REMARKS

Applicant respectfully requests reconsideration of the application.

I. Real party in interest

The real party in interest is the assignee of this application, ATI International SRL, a Society with Restricted Liability chartered in Barbados, West Indies. ATI International is a corporate affiliate of ATI Technologies, Inc. of Toronto, Ontario, Canada.

II. Related Appeals and Interferences

Applicant is unaware of any related appeals or interferences.

III. The status of the claims

Kindly amend the claims as indicated in Exhibit 2. A complete clean copy of all claims involved in the request for reconsideration (as amended during the course of this application), and including those now added by amendment, is attached as Exhibit 1 to this paper.

Claims 1-35, 39-48, and 50-80 are pending in the application. Of these, claims 1, 2, 10, 14, 19, 24, 25, 30, 39, 50, 61 and 70 are independent. Claims 1-35, 39-48, 50-52 and 54-57 are nominally rejected (though as noted below, the Office Action is inadequate to raise any *prima facie* rejection in the manner required by Chapter 2100 of the MPEP, and thus no rejections exist). Claims 36-38 and 49 are cancelled. Claim 53 was omitted inadvertently in the amendment that added claim 54, and is now added by amendment. No rejection of claim 58 was raised in the Action of March 7, 2002. Claims 59-80 are added by amendment.

The claims for which reconsideration is sought are 1-35, 39-48, and 50-57. Initial consideration is requested for claims 58-80.

IV. Status of amendments

Because the application is not under final rejection, amendments proposed in this paper may be entered as of right.

G

The Office Action requests that the first paragraph of the specification be updated to reflect changes in status of the parent applications. None of the parent applications listed on page 1 of the specification have changed status.

A number of claims are amended purely to assist the Examiner in examining the claims. These amendments are not made in response to any statutory rejection, and do not narrow the claims. For example, a number of claims formerly recited "an emulated architecture" and are now amended to recite "the instruction's native architecture."

A number of claims are amended to recite that one action is based "at least in part" on some condition. To the extent that this affects the scope of the claims at all, it broadens them. These amendments are not addressed to any statutory ground of rejection.

V. Telephone interviews of March 7, 2002 and June 26, 2002

Applicant thanks Examiner Eng and Supervisory Examiner Sheikh for an extensive telephone interview of June 26, 2002 and a brief interview on or about March 7, 2002. The following issues were discussed, with agreements as indicated.

A. Brief interview of March 7, 2002

During the last week of February and first week of March, 2002, Applicant left four voice mails for the Examiner and Supervisory Examiner indicating that an interview might be helpful in clearing up any remaining issues of claim scope.

On or about March 7, 2002, Applicant reached the Examiner for a live but brief telephone call. In this brief interview, the Examiner indicated that he understood the claims well enough to examine them and to apply a new reference, and that no extensive interview would be necessary to advance prosecution.

B. Understanding of the claims

Much of the interview of June 26, 2002 was taken up with a survey of the technology. A paraphrase of this overview appears at section VI, at page 10, of this paper.

C. Statements of utility and enablement

In the interview, the Examiner requested indications of statements of utility in the specification. One example appears at page 32, lines 18-20 of the specification:

6

> First, the Tapestry machine exposes both the native RISC instruction set and the X86 instruction set, so that a single program can be coded in both, with freedom to call back and forth between the two. ... Second, an X86 program may be translated into native RISC code, so that X86 programs can exploit many more of the speed opportunities available in a RISC instruction set. This second approach is enabled by profiler **400**, prober **600**, binary translator, and certain features of the memory manager (see sections V through VIII, *infra*).

and another at page 30, lines 20-30:

> Profiler **400** records details of the execution flow of the X86 program. ... Hot spot detector **122** analyzes the profile to find "hot spots," portions of the program that are frequently executed. When a hot spot is detected, a binary translator **124** translates the X86 instructions of the hot spot into optimized native Tapestry code, called "TAXi code." During emulation of the X86 program, prober **600** monitors the program flow for execution of X86 instructions that have been translated into native code. When prober **600** detects that translated native Tapestry code exists corresponding to the X86 code about to be executed, and some additional correctness predicates are satisfied, prober **600** redirects the IP [instruction pointer] to fetch instructions from the translated native code instead of from the X86 code. Probing is discussed in greater detail in [section VI of the specification, pages 100-116].

As discussed in more detail at section IX.D.1.c at page 42, below, these statements are entitled to a presumption of correctness. Any further rejection or requirement relating to these issues must be accompanied by the explanation mandated by MPEP §§ 2164.04, 2164.05, and 2164.08.

D. Suggested amendments to the claims

The Examiner suggested that the word "likelihood" be changed, suggesting that the claims could be phrased in a manner more idiomatic to the ordinary use in the art. The result of this portion of the conversation is discussed in section IX.C.14.c of this paper, at page 39.

The Examiner also suggested several other claim amendments, but in the process of discussing these proposed amendments, it became clear that none was required for novelty, non-obviousness, definiteness or enablement. It was agreed that claims need not "teach" the invention, and that no further rejections would be raised based on such grounds.

Applicant further notes that most of the Examiner's suggestions appear in claims already pending. For example, the idea that two given instructions are drawn from the "same program" is inherent in claim 11 – the input and output of a binary translator, in contexts where execution

G

is transferred between them, constitute "the same program." Cases where "an alternate coding of instructions" does exist are recited in claims 16, 17 and 51.

E. Consideration of the Geppert reference

Agreement was reached that the Geppert reference, submitted in three prior Information Disclosure Statements, would now be considered pursuant to MPEP § 2133.03(b)(IV)(B):

> **B. Nonprior Art Publications Can Be Used as Evidence ...**
> Abstracts identifying a product's vendor containing information useful to potential buyers, . . . along with the date of product release or installation before the inventor's critical date may provide sufficient evidence of prior sale by a third party to support a rejection based on 35 U.S.C. 102(b) or 103. *In re Epstein*, 32 F.3d 1559, 31 USPQ2d 1817 (Fed. Cir. 1994) (Examiner's rejection was based on nonprior art published abstracts which disclosed software products meeting the claims. The abstracts specified software release dates and dates of first installation which were more than 1 year before applicant's filing date.).

See also *GFI Corp. v. Franklin Corp.*, 265 F.3d 1268, 1274, 60 USPQ2d 1141, 1143-44 (Fed. Cir. 2001) ("Materiality is not limited to prior art but instead embraces *any* information that a reasonable examiner would be substantially likely to consider important in deciding whether to allow an application to issue as a patent.") (italic in original, underline added); MPEP § 2128.

A fourth IDS and 1449, and another copy of the Geppert reference, are now enclosed. Applicant requests a checked-off copy of the 1449.

F. "Misleading arguments"

It was agreed (subject to conferring with other Office personnel) that there would be some clarification of the record, to reassure future readers of this prosecution history that no arguments of the Response filed August 10, 2001 were misleading, and that the assertion at page 3, line 18 of the Action of December 2001 is withdrawn. Applicant suggests the following language for inclusion in the Examiner's next paper:

> The Examiner wishes to clarify the record, and acknowledges that Applicant's arguments in the Response filed August 10, 2001 were in no way "misleading" or otherwise improper. The accusation of "misleading argument" made in the Office Action of December 2001 is withdrawn.

G

VI. Summary of the invention

The legal scope of the invention is set forth in the claims; certain specific embodiments are described in the specification. Among other purposes, and without prejudice to the scope of the claims, the invention[1] may be useful to enable a single computer to execute two different instruction set architectures (ISA's), for example, an "old" ISA (*e.g.*, the Intel X86 instruction set) and a "new" ISA (*e.g.*, a RISC instruction set). A single program may mix-and-match routines coded in one ISA with routines coded in the other. The routines may freely call back and forth between ISA's. The following describes the overall contextual setting for the technology, so that the specific inventions of the claims may be more easily understood.

One desirable feature of a reliable two-ISA computer is that the original program text and the control data structures of programs should not be altered by the execution process. For example, if an X86 program uses self-modifying code, any alteration of the X86 instruction text by an emulator might conflict with the self-modification, and would cause the emulation to deviate from the behavior observed if the program were executed on a native X86 computer. Thus, the technique of the Morley '982 patent, discussed in the Office Action of April 2001, might be inoperable in such cases. Similarly, if the X86 segment descriptors are managed by an unmodified X86 program, such as Microsoft Windows, an approach similar to the Richter '684 reference, which modifies system control data structures, may be inoperable, as discussed in section IX.B.5 at page 23 below.

Turning to the specification and Figs. 1a and 1b, Tapestry is fast RISC processor **100**, with hardware and software features that (in addition to the RISC instruction set provided in native mode) provides a correct implementation of an Intel X86-family processor. ("X86" refers to the family including the 8086, 80186, ... 80486, Pentium, and Pentium Pro.) Tapestry processor **100** fetches (stage **110**) instructions from instruction cache (I-cache) **112**, or from memory **118**, from a location specified by IP (instruction pointer, generally known as the PC or

[1] Much of the discussion of the "invention" herein is directed to the entire system disclosed in the application. The term "invention" is used in this paper in its informal sense, and is not intended to be a substitute for the legal definition of the invention set out in the claims. For example, the advantages and results are discussed here only to help establish context for examination, not to further define the legal "invention" of the claims.

G

program counter in other machines) **114**, with virtual-to-physical address translation provided by I-TLB (instruction translation look-aside buffer) **116**. The instructions fetched from I-cache **112** are executed by a RISC execution pipeline **120**. In addition to the services provided by a conventional I-TLB, I-TLB **116** stores several bits **182**, **186** that choose an instruction environment in which to interpret the fetched instruction bytes. One bit **182** selects an instruction set architecture (ISA) for the instructions on a memory page. Thus, the Tapestry hardware can readily execute either native instructions or the instructions of the Intel X86 ISA. This feature is discussed in more detail in section II of the specification, at pages 43-45.

The execution of a program encoded in the X86 ISA is typically slower than execution of the same program that has been compiled into the native Tapestry ISA. Profiler **400** records details of the execution flow of the X86 program. Profiling is discussed in greater detail in section V of the specification, pages 72-100. Hot spot detector **122** analyzes the profile to find "hot spots," portions of the program that are frequently executed. When a hot spot is detected, a binary translator **124** translates the X86 instructions of the hot spot into optimized native Tapestry code, called "TAXi code." The correspondence between X86 code and translated native Tapestry code is maintained in PIPM (Physical Instruction Pointer Map) **602**. During emulation of the X86 program, prober **600** monitors the program flow for execution of X86 instructions that have been translated into native code. When prober **600** detects that translated native Tapestry code exists corresponding to the X86 code about to be executed, and some additional correctness predicates are satisfied, prober **600** redirects the IP to fetch instructions from the translated native code instead of from the X86 code. That is, as the X86 program is executed, and control reaches a portion of the X86 program that has been translated, the computer will automatically recognize that a RISC alternative coding exists, and will transfer control to the version translated into the RISC ISA. Probing is discussed in greater detail in section VI of the specification, at pages 100-116.

The Tapestry machine exposes both the native RISC instruction set and the X86 instruction set, so that a single program can be coded in both, with freedom to call back and forth between the two. This approach is enabled by ISA bit **180**, **182** control on converter **136**, or in an alternative embodiment, by ISA bit **180**, **182**, calling convention bit **200**, semantic context record **206**, and the corresponding exception handlers (see section IV of the specification, pages

63-72). Second, an X86 program (or portions thereof) may be translated into native RISC code, so that X86 programs can exploit many more of the speed opportunities available in a RISC instruction set. This second approach is enabled by profiler **400**, prober **600**, binary translator, and certain features of the memory manager (see sections V through VIII of the specification, pages 72-116) Third, these two approaches cooperate to provide an additional level of benefit.

Some aspects of the invention relate to several different "side tables" that are used to implement these features.

In one particular embodiment that is the subject of several of the claims, a side table (PFAT **172**) has entries **174** that correspond to regions of X86 instructions. The probe bits **624** of PFAT entries **174** give an approximate indication, a likelihood estimate, of whether there is a RISC translated code segment for any of the code in the region corresponding to the PFAT entry.[2] Another side table (PIPM **602**) is consulted when the PFAT table access indicates that existence of translated code is likely. The PIPM then gives the final definitive answer of whether execution should be transferred to RISC code, and where the relevant RISC code is located in memory. This embodiment is the focus of **Figs. 6a-6c** and section VI (pages 100-116).

In another example embodiment, ISA bits **180**, **182**, **194** each correspond to a region of memory, telling whether instructions in the region are to be interpreted in the X86 ISA or in the RISC ISA. When execution flows from a region whose ISA bit indicates one ISA to a region indicating the other, the computer reconfigures itself so that execution may resume in the new ISA, and so that the result of the mixed-ISA computation will be the same as the result that would be obtained if the program were coded in a single ISA. (In one embodiment, ISA bits **180** are stored in the same physical PFAT **172** as the probe bits **624**. In other embodiments, these two conceptually-distinct sets of bits could be stored in distinct storage structures. For example, copies **182** of the ISA bits are cached in the instruction translation look-aside buffer (I-TLB) **116**, and may be stored with or separately from the probe bits **624**). This embodiment is the focus of **Figs. 3a-3o** and section II of the specification (pages 44-46).

[2] In some embodiments, the ISA bits 180 and probe bits 624 may be stored together in a single table. In others, they might be stored separately.

A third embodiment includes a "calling convention" (CC) bit **196, 200**. In any computer, data that are to be passed from one routine of a program to another routine are passed according to a "calling convention," an agreement among software components for how data are to be passed from one component to the next. A given system's calling convention is typically defined by the designers of the computer system or by designers of compilers for the system. The CC bit **196, 200** of PFAT **172** and I-TLB **116**, is disclosed in **Figs. 2a-2c** and section IV (pages 64-73).

The TAXi system uses interrupts (i) to seize control of the execution of a program, (ii) to transfer to control from the X86 code to the translated RISC code, (iii) to alter machine state so that coding assumptions embodied in one code segment are rendered consistent with coding assumptions of a segment to which control is transferred, and then (iv) to transfer control back to the appropriate point in the X86 code at the end of the translated segment. The TAXi system handles conventional interrupt-driven functions like page faults or other synchronous or asynchronous execution interrupts. In addition, the TAXi system adds a number of unconventional functions like changing the ISA in which the next instructions will be executed, and many of these functions are performed by interrupts. Some interrupts are generated by instructions like TRAP, that are architecturally-defined to generate interrupts. In the TAXi system, some interrupts are generated on simple instructions like ADD's or JUMP's that are not architecturally-defined to generate an interrupt. For example, if a translated hot spot starts with a simple integer ADD instruction (the definition in the X86 architecture of an integer ADD instruction does not call for an interrupt to be raised), the side tables are set to values that cause this normally-interrupt-free instruction to raise an interrupt, so that the TAXi system can gain control and transfer execution over to the translated RISC code. Some control transfer instructions that are architecturally-defined to transfer control to one address are altered, so that control is transferred to a destination other than the architecturally-defined destination.

Many of the claims of this application relate to techniques for (i) recognizing when X86 execution has reached a hot spot for which a translation exists, when the instruction text itself and control data structures should not be changed, (ii) recognizing when the hardware should switch modes from executing in X86 mode to RISC mode, or vice-versa, and then (iii) transferring control back to the X86 code at the point corresponding to the end of the translated hot spot, all without altering the original X86 program text.

G

Many of the claims of this application are directed to computer design techniques that have broad applicability beyond the specific context described above. Thus, unless a claim specifically recites "two distinct ISA's," "emulation of a non-native ISA," "raising an interrupt," etc. it should be understood that the claim is not so limited. The discussion in this preliminary section is merely to provide context for understanding the technology, not to state the scope of the claims. Therefore, it would not be helpful to examine the application based on the above description; the focus of examination should remain on the claims themselves.

VII. Issues presented for reconsideration

The following issues are presented for reconsideration:

a. whether any enablement rejection under 35 U.S.C. § 112 ¶ 1 has been raised against claims 1-35, 39-48, and 50-57.

b. whether any vagueness rejection under 35 U.S.C. § 112 ¶ 2 has been raised against claims 1-35, 39-48, and 50-57.

c. whether any obviousness rejection under 35 U.S.C. § 103 has been raised against claims 1-35, 39-48, and 50-57 based on U.S. Patent No. 5,481,684 to Richter.

VIII. Grouping of claims

The claims are grouped in a number of groups according to individual issues. The groups are defined, and stand or fall together or separately, as discussed in section IX, below.

IX. Argument

A. Preliminary statement

Applicant assures the Examiner – <u>the claims mean what they say</u>. With only one exception ("data manipulation behavior," see section IX.C.4, below), the claims are drafted using only conventional terms of art and other recognized idioms, in their commonly-used sense. Applicant requests that the Examiner read the claims carefully, and examine the claims as they are stated. The Examiner should carefully avoid paraphrasing the claims (it appears that several of the "rejections" in the Action of March 2002 are based purely on misquotation of the claims, and not on any defect in the claims themselves), or otherwise straying from the exact words of the claims. Surprise at the unconventional features of the claims should not be turned into

G

rejections for vagueness or non-enablement. Similarly, the surprising combination of elements in the claims suggest non-obviousness, not obviousness.

B. Obviousness

Claims 1-52 and 54-57 are nominally rejected under 35 U.S.C. §103 over U.S. Patent No. 5,481,684 to Richter.

1. Group I: claims 1-9, 17, 23, 34, 35, 41, 50-56, and 78-80

The Action of March 2002 purports to reject claim 2 over the Richter '684 patent. Claim 2 may be considered as a representative claim of a group that includes claims 1-9, 17, 23, 34, 35, 41, 50-56, and 78-80 (Group I): if claim 2 stands, the rest of Group I stands with claim 2. If claim 2 falls, then the other claims must be considered separately to the extent discussed elsewhere in this Response. Claim 2 recites as follows:

> 2. A method, comprising the steps of:
>
> as part of the basic instruction cycle of executing an instruction of a non-supervisor mode program executing on a computer, consulting a table, the table having entries that are indexed by the address within an address space of instructions executed, entries of the table containing attributes of instructions whose addresses index to the respective table entries; and
>
> controlling an architecturally-visible data manipulation behavior or control transfer behavior of the instruction based at least in part on a content of a table entry indexed by the address of the instruction.

a. First ground of traverse: Richter '684 does not teach a table with entries "indexed by an address within an address space"

Claim 2 recites a table "indexed by the address within an address space" of instructions fetched from the address space.

At best, **Richter '684 shows** circuitry that alters the behavior of instructions based on the address space in which instructions are located, not an address "within the address space."[3]

[3] The references to the claim and to Richter '684 are bolded to assure the Examiner that Applicant is not "merely pointing out what the claims require," but is instead "identifying a difference between the references and the claims and explaining how the claims are patentable over the prior art."

These contrasts have been drawn in Applicant's previous papers, though the remarks in the Office Action suggest that these contrasts may have been overlooked.

G

To understand this distinction, it is helpful to recollect that when a program executes on an Intel X86, a program references several distinct segments. Each segment is a separate address space, each of which is defined by a segment register. An Intel X86 program typically has one or more code segments, one or more data segments, and a stack segment. (See Intel manual, vol. 1, Exhibit 5, pages 3-7 to 3-10; Intel manual, vol. 3, Exhibit 7, pages 3-3 to 3-24, 3-33 to 3-34 and 4-12 to 4-14.) Each of these segments defines a different address space. Thus, address 0 in a first code segment may be a different memory location than address 0 in a second code segment, and each may be different than address 0 in any of the data segments, which may all be different than address 0 in the stack segment.[4] This is a familiar notion – in essentially all computers made since the Intel 80286 and Motorola 68010 were introduced in about 1985 (and all large computers since about 1970), different programs have different address spaces – so that a write by program A into its address 20, for example, does not conflict with the data that program B has stored at its address 20. Intel takes this one step further, so that the program text, the data and the stack for a single program may be stored in different address spaces – a store into the stack segment is protected from data stored in the code and data segments. In the Intel X86 architecture, each memory reference must designate a particular segment. For example, most branch instructions implicitly refer to addresses in the current code and data segment. PUSH and POP instructions implicitly refer to the current stack segment. Most MOVE instructions implicitly refer to the current data segment. A few instructions explicitly designate a particular segment.

In light of that technological background, the **distinction** between the "table indexed by an address within an address space" **of the claim** and the segment-register based scheme of **Richter '684** becomes clear. **Richter '684** stores a bit combination in an Intel-like segment descriptor (col. 8, line 65 to col. 9, line 2; col. 9, lines 28-31; col. 10, lines 35-44) that designates whether an instruction is to be interpreted in the Intel X86 ISA or in the IBM/Motorola PowerPC ISA. **Richter's** bit combination is effective for the entire segment, that is, for all addresses within the address space designated by the descriptor. **Richter '684** provides no way to specify

[4] It may also be the same memory locations, in some cases – two or more segments may map to the same origin in the linear address space. However, Richter does not discuss this special case, and thus it is not inherent in Richter.

G

that two different addresses "within an address space" are to "index" to two different table entries, as recited **in the claim**.

Because claim 2 recites a limitation absent from Richter '684, claim 2 is not obvious over Richter '684.

b. Second ground of traverse: the written Office Action is inadequate to raise a *prima facie* rejection

There is no obviousness rejection of these claims, because the Office Action fails to make the minimum showings required for a rejection to exist.

Ever since it was initially filed, claim 2 has recited "a table indexed by an address of an instruction." This language has been distinctly pointed out in each of Applicant's three previous papers, and contrasted to the prior art (Response of February 4, 2002 at page 4-5; Response of August 10, 2001 at page 23, lines 23-25; Response of October 10, 2000 at page 11, lines 8-9). Two prior art references have been withdrawn based on this language. This language is not questioned under § 112 – there appears to be no possible explanation for the omission.

MPEP § 2143.03 reads as follows (italic in original, underline added):

> **2143.03 All Claim Limitations Must Be Taught or Suggested**
> To establish *prima facie* obviousness of a claimed invention, all the claim limitations must be taught or suggested by the prior art. *In re Royka*, 490 F.2d 981, 180 USPQ 580 (CCPA 1974). "All words in a claim must be considered in judging the patentability of that claim against the prior art." *In re Wilson*, 424 F.2d 1382, 1385, 165 USPQ 494, 496 (CCPA 1970). ....

Because claim 2 recites a limitation that is not considered by the § 103 portion of the Office Action, no § 103 rejection exists.[5] Because no rejection has been raised, no amendment is made in response to any § 103 rejection.

c. All claims of Group I stand with claim 2

For reasons discussed in section IX.B.1.b, there is no § 103 rejection of claim 2, or of any other claim of Group I. As discussed in section IX.B.1.a, any attempt to reject claim 2 over Richter '684 would fail on the merits. The claims of Group I are not rejected, and are allowable.

[5] Because no obviousness rejection exists, any rejection in the next Office Action will either be insufficient to be a rejection at all, or will be a "new ground of rejection" – in either case, no such future rejection can be made final.

2. Group II: claims 1, 7, 12, 14-18, 24-29, 42, 57, 60, 66 and 73

The Action of March 2002 purports to reject claim 14 over Richter '684. Claim 14 may be considered as a representative claim of a group that includes claims 1, 7, 12, 14-18, 24-29, 42, 57, 60, 66 and 73 (Group II): if claim 14 stands, the rest of Group II stands with claim 14. If claim 14 falls, then these other claims must be considered separately to the extent discussed elsewhere in this Response.

Claim 14 recites as follows:

> 14. A microprocessor chip, comprising:
> instruction pipeline circuitry;
> address translation circuitry; and
> a lookup structure having entries associated with corresponding address ranges generated by the instruction pipeline circuitry and translated by the address translation circuitry, the entries describing a likelihood of the existence of an alternate coding of instructions located in the respective corresponding address range.

a. First ground of traverse: Richter '684 does not teach "a likelihood of the existence of an alternate coding of instructions"

Claim 14 recites a lookup structure with an entry that describes "a likelihood of the existence of an alternate coding of instructions."

In contrast, **Richter '684** teaches nothing remotely analogous. At best, the portions of **Richter '684** indicated by the Office Action teach that different pieces of a program can be coded in two different ISA's. *E.g.*, col. 2, lines 38-40; col. 5, lines 55-67. The indicated portions of **Richter** '684 indicated in the Office Action never indicate that a single program segment might exist in two alternate codings. Without an "alternate coding," a table that indicates "a likelihood of the existence of an alternate coding" (as recited in **claim 14**) cannot possibly be suggested by **Richter '684**.

b. Second ground of traverse: the written Office Action is inadequate to raise a *prima facie* rejection

There is no obviousness rejection of these claims, because the Office Action fails to make the minimum showings required for a rejection to exist.

G

Even in its initially filed form, claim 14 recited a lookup structure whose entries "describe a likelihood of the existence of an alternate coding of instructions." In the past, this language has been indicated to be comprehensible – note that there was no objection to this language in the Action of December 2001, and claim 14 has been allowed over two other references based on this language. All three of Applicant's prior papers have drawn specific attention to this language. Applicant suggests that it is improper for this language to now be disregarded in a fourth Office Action.

MPEP § 2143.03 reads as follows (bold and italic in original, citations omitted, underline added):

> **2143.03 All Claim Limitations Must Be Taught or Suggested**
> To establish *prima facie* obviousness of a claimed invention, all the claim limitations must be taught or suggested by the prior art. All words in a claim must be considered in judging the patentability of that claim against the prior art....
>
> **INDEFINITE LIMITATIONS MUST BE CONSIDERED**
>
> A claim limitation which is considered indefinite cannot be disregarded. If a claim is subject to more than one interpretation, at least one of which would render the claim unpatentable over the prior art, the examiner should reject the claim as indefinite under 35 U.S.C. 112, second paragraph (see MPEP § 706.03(d)) and should reject the claim over the prior art based on the interpretation of the claim that renders the prior art applicable. ...
>
> **LIMITATIONS WHICH DO NOT FIND SUPPORT IN THE ORIGINAL SPECIFICATION MUST BE CONSIDERED**
>
> When evaluating claims for obviousness under 35 U.S.C. 103, all the limitations of the claims must be considered and given weight, including limitations which do not find support in the specification as originally filed (i.e., new matter). ... [It] was error to disregard [claim limitations that did not appear in the specification as filed] when determining whether the claimed invention would have been obvious in view of the prior art.

In contrast, the Office Action states (page 6, lines 17-19):

> For the reasons set forth in the section 112 rejections above, no statement can be made as to whether the Richter references meets the functional languages of the claims ...

The Office Action is in irreconcilable conflict with the MPEP. MPEP § 2143.03 is unambiguous – it is examiner error to disregard claim limitations merely because the claim language is subject to a rejection under § 112 ¶¶ 1 or 2. If a claim limitation is thought indefinite or unsupported, MPEP § 2143.03 requires an examiner to make a good faith "best guess" as to

G

the broadest reasonable "interpretation of the claim," and compare the prior art against that "best guess." It is impermissible to disregard the claim limitation entirely, as the Examiner confesses to have done here.

Similarly, MPEP § 2143.03 makes no exception for "functional" language in an obviousness rejection. Such distinctions were made some decades ago in apparatus claims, but the law has changed, and no longer allows different treatment of "structural" and "functional" language (except in the context of § 112 ¶ 6 limitations – not an issue here). Further, there has never been any proper basis to do what has been done here – disregard "functional" limitations in method claims.

Applicant is unaware of any PTO regulation that gives an examiner the authority to ignore an instruction of the Director and Commissioner ordered through the MPEP. Applicant is similarly unaware of any subsequent Order or Notice that would supersede MPEP § 2143.03. If the Examiner is aware of either, he is requested to provide a copy. Unless the Examiner can supply such, Applicant suggests that compliance with MPEP § 2143.03 would be in order.

Finally, it is well-established that where an agency employee acts in "brazen defiance" of agency regulations, that employee's action has no legal existence. *Mayor and City Council of Baltimore v. Mathews*, 562 F.2d 914, 920 (4th Cir. 1977); see also *Certain Former CSA Employees v. Dept. of Health and Human Services*, 762 F.2d 978, 984 (Fed. Cir. 1985) (action in violation of agency's own regulation is "illegal and of no effect"). Similarly, the absence of required findings is fatal to the validity of the Office Action, regardless of whether there may be evidence in the record to support proper findings. *Anglo-Canadian Shipping Co. v Federal Maritime Com.*, 310 F.2d 606, 617 (9th Cir. 1962). In view of MPEP § 2143.03, a written rejection that omits claim limitations has no legal existence.

3. Group III: Claims 1, 18-23, 30-34, 43, 58, 59, 67 and 78

The Action of March 2002 purports to reject claim 19 over the Richter '684 patent. Claim 19 may be considered as a representative claim of a group that includes claims 1, 18-23, 30-34, 43, 58, 59, 67 and 78 (Group III): if claim 19 stands, the rest of Group III stands with claim 19. If claim 19 falls, then these other claims must be considered separately to the extent discussed elsewhere in this Response.

Claim 19 recites as follows:

> 19. A microprocessor chip, comprising:
> instruction pipeline circuitry; and
> interrupt circuitry cooperatively designed with the instruction pipeline circuitry to trigger a synchronous interrupt on execution of an instruction of a process based at least in part on a memory state of the computer and the address of the instruction, wherein the architectural definition of the instruction in the instruction's native architecture does not call for an interrupt.

a. First ground of traverse: Richter '684 does not disclose "triggering an interrupt ... wherein the architectural definition of the instruction in the instruction's native architecture does not call for an interrupt"

Claim 19 recites "interrupt circuitry ... to trigger an interrupt on execution of an instruction ... wherein the architectural definition of the instruction in the instruction's native architecture does not call for an interrupt."

The portions of **Richter '684** indicated by the Office Action disclose exactly the opposite of the claim. For example, **Richter '684** states that he sets certain bits in a segment descriptor to "an invalid or reserved combination of bits" (col. 8, lines 65-66). **Richter '684** states that this setting "could cause a prior-art x86 system to perform an undocumented function" (col. 9, lines 4-6), and that **Richter's** system gains control when "an unknown opcode is detected by instruction decode" (col. 12, lines 4-5). The architectural definition of the X86 calls for an interrupt when the processor executes an "undocumented function" or "unknown opcode." (See Intel manual, vol. 1, Exhibit 5, page 4-12; Intel manual, vol. 2, Exhibit 6, pages A-5 n.1 and A-7 n.1; Intel manual, vol. 3, Exhibit 7, page 17-5.) Thus, these portions of **Richter '684** cannot meet a claim limitation "wherein the architectural definition of the instruction ... does not call for an interrupt."

Because claim 19 recites a limitation that is absent from Richter '684, claim 19 is non-obvious over Richter '684.

b. Second ground of traverse: the Office Action fails to consider each limitation of the claims, in violation of MPEP § 2143.03

For reasons discussed in section IX.B.2.b, the Office Action fails to state an obviousness rejection of any claim in Group III. For example, the language "[triggering] an interrupt on execution of an instruction ... the architectural definition of the instruction not calling for an

G

interrupt" has been part of claim 19 since its original filing, and has been pointed out in each of Applicant's papers, yet never considered in the § 103 portion of any Office Action.

Because no Office Action has made any attempt to reject any claim of Group III over the prior art, no such rejection has never existed.

4. Group IV: Claims 1, 10-13, 26, 35, 39-48, 68 and 74

The Action of March 2002 purports to reject claim 10 over Richter '684. Claim 10 may be considered as a representative of a group that includes claims 1, 10-13, 26, 35, 39-48, 68 and 74 (Group IV): if claim 10 stands, the rest of Group IV stands with claim 10. If claim 10 falls, then these other claims must be considered separately to the extent discussed elsewhere in this Response.

Claim 10 recites as follows:

> 10. A microprocessor chip, comprising:
>
> instruction pipeline circuitry;
>
> table lookup circuitry designed to index into a table by a memory address of a memory reference arising during execution of an architecturally-defined instruction, and to retrieve a table entry corresponding to the address, the table entry being distinct from the memory referenced by the memory reference;
>
> the instruction pipeline circuitry being responsive to the contents of the table entry to alter a manipulation of data or control transfer behavior of the instruction in a manner incompatible with the architectural definition of the instruction in the instruction's native architecture.

a. First ground of traverse: Richter '684 does not teach altering any instruction's behavior "in a manner incompatible with the architectural definition of the instruction"

Claim 10 recites "[altering] a … behavior of the instruction in a manner incompatible with the architectural definition of the instruction."

In contrast, **Richter '684** teaches exactly the opposite. **Richter '684** teaches switching ISA's so that every instruction is executed in a manner to improve compatibility of the execution of each instruction with the architectural definition of the instruction in the instruction's own architecture. *See, e.g.*, col. 2, lines 34-43; col. 5, lines 54-67; col. 10, lines 34-45.

G

b. Second ground of traverse: the Office Action fails to consider each limitation of the claims, as required by MPEP § 2143.03

For reasons discussed in section IX.B.2.b, the Office Action fails to state an obviousness rejection of any claim in Group IV. For example, since its original filing, claim 10 has recited "[indexing] into a table by a memory address of a memory reference arising during execution of an instruction" or similar language. This limitation has never been considered over the art in any Office Action.

Because no Office Action has made any attempt to reject any claim of Group IV over the prior art, no such rejection has never existed.

5. Group V: Claims 1, 53, and 59-77

Claims 59-77 are added to claim a new aspect of the invention, related to the aspect claimed in Group I (claims 2 and 50). Claim 70 may be considered as a representative claim of a group that includes claims 1, 53, and 59-77 (Group V). If claim 70 stands, the rest of Group V stands with claim 70. If claim 70 falls, then these other claims must be considered separately to the extent discussed elsewhere in this Response.

Claim 70 recites as follows:

> 70. An apparatus, comprising:
>
> instruction pipeline circuitry; and
>
> table lookup circuitry designed to retrieve a table entry from a table whose entries are indexed by an address of an instruction fetched for execution, the table being stored in storage that is architecturally invisible to programs in the fetched instruction's native architecture;
>
> the instruction pipeline circuitry being responsive to a content of the table entry to control an architecturally-visible data manipulation behavior or control transfer behavior of the fetched instruction based at least in part on a content of the table entry associated with the address of the fetched instruction..

Claim 70 recites a table that is stored "in storage that is architecturally invisible to programs."

In contrast, **Richter's** "instruction set type bit 21" is stored in the x86 segment descriptors and segment registers (col. 10, lines 34-37). The segment descriptors and segment registers are architecturally visible in the x86 architecture. (See Intel manual, vol. 1, Exhibit 5, pages 3-7 to 3-9; Richter '684, col. 9, lines 4-5).

G

Because claim 70 recites a limitation that is absent from Richter '684, claim 70 is non-obvious.

6. Dependent claims

The pre-existing dependent claims, 3-9, 11-13, 15-18, 20-23, 26-29, 31-35, 40-48, 51, 52, and 54, are purportedly rejected over the art. However, the written Office Action does not address the limitations recited in these claims. Such piecemeal examination is discouraged by 37 C.F.R. § 1.105 and MPEP § 707.07(g). In particular, the following claim limitations are not discussed in the § 103 section of the Office Action:

transfer of execution control to a second instruction for execution	claims 3, 44
transfer of execution control to an instruction coded in an instruction set architecture (ISA) different than the ISA of the executed instruction	claim 4
entries of the table correspond to pages managed by a virtual memory manager	claims 8, 27, 48, 52-55
the table entries are indexed by a physical address	claim 56
circuitry for locating an entry of the table is integrated with virtual memory address translation circuitry of the computer	claims 8, 27
triggering an interrupt on execution of an instruction ... based at least in part on ... the address of the instruction	claims 9, 12, 18, 29, 43
a binary translator	claim 11
pipeline control circuitry ... designed to initiate a determination of whether to transfer control from an execution of the instruction ... to the second binary representation	claim 11
returning control to an instruction flow of the process other than the instruction flow triggering the interrupt	claims 13, 20, 31
the table entry is an entry of a translation look-aside buffer	claim 15
two different instruction flows that are logically equivalent to each other	claims 22, 33
consulting a second table, the entries of the second table definitively indicating entry points for initiating such alternate codings as exist	claim 57

(Many of these limitations are not discussed in the § 112 ¶ 1 or ¶ 2 context, so even under the view of the law espoused in the Office Action and discussed in section IX.B.2.b, the failure to discuss these claims is inexplicable.) Because there has been no attempt to comply with the minimum requirements for raising an obviousness rejection of these claims, no such rejection exists.

C. Indefiniteness

1. Legal infirmity of any "indefiniteness" rejections

All "indefiniteness" rejections raised in this action are suspect. First, very nearly every issue has already been raised and resolved earlier in prosecution. Second, nearly every term used in the claims is a well-established term of art, used in its conventional sense – there is no "indefiniteness" in describing the invention in these terms. Third, the "indefiniteness" portions of the Action attempt to apply rules that simply do not exist.

a. Most of the "indefiniteness" concerns have been raised and resolved earlier in prosecution

Very nearly all of the nominal rejections raised in this Office Action under § 112 ¶ 2 have been previously raised and resolved to the Examiner's satisfaction. For example, the Office Action of April 10, 2001 queried about the phrase "architecturally-visible data manipulation behavior," and Applicant provided an explanation in the Response of August 10, 2001, at page 15 (see section IX.C.4 at page 28, below, for an amplification of this explanation). This explanation was accepted by the Examiner, as indicated by the fact that the query was dropped from the Office Action of December 4, 2001.

Nonetheless, in order to demonstrate a good faith effort to advance prosecution, Applicant will go well beyond the minimum legal requirements for establishing patentability, and offer some explanatory assistance for claim language that has not been discussed earlier in prosecution. It should be understood that this explanation is provided as a convenient concrete frame of reference to assist examination of the application, and does not limit the claims to only the embodiments described here.

b. Erroneous tests for "definiteness"

(i) "Definitions in the specification"

A number of the purported § 112 ¶ 2 rejections state that the basis for the rejection is because "The Examiner is unable to find their definitions in the specification." These claims have been drafted in reliance on MPEP § 2111.01, which permits an applicant to <u>not</u> to define claim terms in the specification, and instead rely on the common definition in the art. MPEP § 2111.01 instructs the Examiner that "When not defined by applicant in the specification, the words of a claim must be given their plain meaning. In other words, they must be read as they would be interpreted by those of ordinary skill in the art."

Most of the § 112 ¶¶ 1 and 2 issues raised in the Office Action relate to established terms of art, used in the claims in their conventional senses. Because neither § 112 ¶ 2 nor the MPEP provide basis for rejecting established terms of art under § 112 ¶ 2, there are no "rejections" at all. However, in an effort to usefully advance prosecution, these claim terms are discussed below. If these issues are raised again, the Examiner is requested to identify a provision of the MPEP that creates some exception to § 2111.01. The Examiner is requested not to raise rejections or requirements that are not authorized.

(ii) "Support for" or "meaningful operation"

Several of the purported § 112 ¶ 2 rejections use phrases such as "support for" or "meaningful operation" that might be relevant under § 112 ¶ 1, but are inapplicable to any known ground of rejection arising under § 112 ¶ 2. MPEP § 2174 makes clear that ¶ 1 and ¶ 2 impose distinct requirements, and that it is improper to pose a rejection under paragraph two using reasoning that is only applicable in the context of paragraph one. In *Carl Zeiss Stiftung v. Renishaw PLC*, 945 F.2d 1173, 1180-81, 20 USPQ2d 1094, 1100 (Fed. Cir. 1991), the Federal Circuit held that there is no requirement that a claim recite every component required for an "operable" device. These nominal "rejections" cannot be considered "rejections" at all.

2. "Each entry describing a likelihood of the existence of an alternate coding of instructions"

The Office Action begins as follows:

> With respect to all independent claims, the recitation "each entry describing a likelihood of the existence of an alternate coding of instructions" is vague and indefinite. It is not clear whether there is or there is no alternate coding of instructions in the system for causing the pipeline to behave differently. Note that the definition of "likelihood" is "probability" in Webster's New Collegiate Dictionary. The probability of existence (instead of "true" or "false") would not result in two outcomes for designating two different behaviors. Note further that a pipeline which is a digital circuit is able to respond to only precise instructions and not probability.

This paragraph raises a significant number of subsidiary issues that can best be handled one-by-one. These subsidiary issues are discussed in the following sections, and then the remaining issues in the paragraph are considered in section IX.C.14 at page 37, below.

3. "Architectural definition of an instruction"

The Office Action states:

G

> The scope of the meaning of the following is not clear:
>
> 1. "the architectural definition of the instructions"... The Examiner is unable to find their definitions in the specification.

a. This issue has been raised and resolved earlier in prosecution

This issue appears to have been raised in error. An almost identical issue was raised in the Office Action of April 10, 2001, and the following response was provided in the Response of August 10, 2001, at pages 15 and 17:

> **K. "alter—in manner incompatible with the architectural definition of the instruction"**
>
> The Examiner requested information as to the meaning of "alter — in manner incompatible with the architectural definition of the instruction" as used in claim 10.
>
> As is well known in the art, an architecture defines the behavior of each instruction in its instruction set. For example, most architectures define an "ADD" instruction to cause the addition of two numbers, a "JUMP" instruction to transfer control to another instruction in accordance with defined rules, and similar definitions for all other instructions in the instruction set. ...
>
> Applicant believes that the plain language of the clause would be will understood by those in the art.

This was apparently accepted as a sufficient explanation, because the issue was dropped from the Office Action of December 4, 2001.

b. § 112 ¶ 2 does not require explicit definitions of well-established terms of art

The phrase "architectural definition of an instruction" is an established term of art. For example, the concept appeared in undergraduate textbooks nearly 20 years ago (Tanenbaum, Exhibit 3, pages 181-82) and has been used many times since (Hennessey & Patterson, Exhibit 4, pages 89-92; Intel manual, vol. 2, Exhibit 6, pages A-1 to A-7). An "architectural definition of an instruction" is the description of an instruction's behavior in the relevant computer's architectural definition. There is no need to provide a definition of established terms of art in the specification.

G

4. "an architecturally-visible data manipulation behavior"

The Office Action states:

> The scope of the meaning of the following is not clear:
>
> 1. ... "an architecturally-visible data manipulation behavior" ... The Examiner is unable to find their definitions in the specification.

a. This issue has been raised and resolved earlier in prosecution

This issue appears to have been raised in error. An almost identical issue was raised in the Office Action of April 10, 2001, and the following response was provided in the Response of August 10, 2001, at page 15:

> **F. "architecturally-visible behavior"**
>
> The Examiner requested information as to the meaning of "an architecturally-visible data manipulation behavior or control transfer behavior of an instruction" as used in claim 1.
>
> "Architecturally-visible data manipulation behaviors or control transfer behaviors of an instruction" are discussed in the documents incorporated by reference into the specification at pages 138-139. As is well-known in the art, "architecturally visible behaviors" are behaviors that must be preserved across all implementations of an architecture. For example, the bit sequence "0000 0100" means "add an 8-bit immediate to the A register" in all implementations of the x86 architecture, from the 8086 in the mid-1970's to the Pentium III. On the other hand, pipeline hazard controls, the internal sequencing of a repeated string instruction, and the management of hardware resources during intermediate pipeline stages are not architecturally visible in most architectures. Most architecturally-visible results persist across instruction boundaries, most non-architecturally-visible results do not. [6]

[6] "Architecturally visible" is a well-established term of art, as demonstrated by the industry papers and university course notes, at Exhibit 8.

Another "rule of thumb" for distinguishing "architecturally-invisible" and "architecturally-visible" behaviors is that an element is generally considered "architecturally invisible" if no program can be written that can tell the difference between two different states of the element, and "architecturally visible" if a program can tell the difference. For example, a memory location that cannot be addressed, and that has no effect on the execution of any instruction, is "architecturally invisible." As another example, the difference between a hardware TLB fill and a software TLB fill is "architecturally invisible" at the application level, and "architecturally visible" to the operating system. See Tanenbaum, Exhibit 3, p. 181. The quality of an emulation system is reflected, at least partially, in the degree to which the emulation system itself is architecturally invisible to the emulated program. In any particular context, engineers in the art have no difficulty distinguishing "architecturally visible" and "architecturally invisible" components or behaviors from each other, and continually rely on clear and definite (though context-dependent) notions of architectural visibility and invisibility in designing computers.

G

> Some instructions have "architecturally-visible data manipulation behavior" – for example, an ADD instruction calls for an addition of data, and all implementations of the architecture must achieve the same result. Other simple data manipulations include subtraction, multiplication, division, negation, logical and, logical or, logical exclusive or, complement, shift, bit field insert or extract, most format conversions, and most floating-point arithmetic operations, etc. ("Data manipulation" behavior can be contrasted to "data movement" behavior of a memory load or store)
>
> . . .
>
> Applicant believes that these concepts are well known in the art and that no amendments are necessary to address the issues raised by the Examiner.
>
> Some examples of controlling architecturally-visible instruction behavior based on table entries are discussed in section VI.A (page 100-101 of the specification), section VI.B (pages 101-103), and section VI.D (pages 106-111). The specification may include other examples.

This was apparently accepted as a sufficient explanation, because the issue was dropped from the Office Action of December 4, 2001.

Additional examples of "controlling an architecturally-visible data manipulation behavior or control transfer behavior" appear at Fig. 6c, specification at pages 110; at section II (pages 44-46), and section VI.E (pages 111-112).

Section IV of the specification (pages 64-73) discusses an embodiment in which a side table may alter the behavior of a routine data-manipulation instruction, an instruction that generates, as a primary output, a bit pattern that was not present in any of its inputs. Examples include add, subtract, logical or, logical and, floating-point-to-integer convert, or similar data-manipulation instructions. In section IV, either the ISA or calling convention bit may cause such an instruction to move data from a block of registers to memory or from memory into the register block, or may cause the hardware to convert from processing the X86 ISA to a RISC ISA.

5. "control transfer behavior"

The Office Action states:

> The scope of the meaning of the following is not clear:
>
> 1. … "control transfer behavior" in all the independent claims. The Examiner is unable to find their definitions in the specification.

a. This issue has been raised and resolved earlier in prosecution

This issue appears to have been raised in error. An almost identical issue was raised in the Office Action of April 10, 2001, and the following response was provided in the Response of August 10, 2001, at page 15:

> An instruction may have a "architecturally-visible control transfer behavior" – for example, a JUMP instruction calls for a control transfer to a particular program location, and all implementations of the architecture must achieve the same result.

This was apparently accepted as a sufficient explanation, because the issue was dropped from the Office Action of December 4, 2001.

b. § 112 ¶ 2 rejection of a well-established term of art is unwarranted

"Control transfer" is an established term of art, long known in undergraduate curricula. See, for example, Hennessey & Patterson, Exhibit 4, pages 103-109; Intel manual, vol. 2, Exhibit 6, pages 3-245 to 3-251; and the exhibits included with the Response of October 2000. No § 112 ¶ 2 rejection is warranted.

6. "wherein the architectural definition of the instruction in an emulated architecture does not call for an interrupt"

The Office Action states:

> The scope of the meaning of the following is not clear:
>
> . . .
>
> 2. "wherein the architectural definition of the instruction in an emulated architecture does not call for an interrupt" of claims 19 and 30. The Examiner is unable to find the explanation of the wherein clause in the specification.

a. An incorrect legal test is applied

Section 112 ¶ 2 requires only that the <u>claims</u> be clear and definite, not that the <u>specification</u> elaborate every claim limitation. Because an incorrect legal test has been applied, no rejection exists.

b. This issue has been raised and resolved earlier in prosecution

This issue appears to have been raised in error. An almost identical issue was raised in the Office Action of April 10, 2001, and the following response was provided in the Response of August 10, 2001, at pages 14 and 21:

> **E. "architectural definition ... not calling for an interrupt"**
>
> The Examiner requested information as to the meaning of "the architectural definition of the instruction not calling for an interrupt" as used in claim 1.
>
> An instruction "wherein the architectural definition of the instruction ... does not call for an interrupt" is an instruction that the architecture defines as executing without raising an interrupt. For example, in most computers, the definition of an integer add of two integer registers, where one register contains "2" and the other contains "3," does not call for an interrupt. On the other hand, as is well-known in the art, the definition of an SVC or TRAP instruction calls for an interrupt. In most architectures, a divide instruction, in cases where the divisor is zero, calls for an interrupt. Most architectures define that an access to an undefined memory location calls for an interrupt. Applicant believes that this concept is well known to those skilled in the art.
>
> Some examples of an instruction "wherein the architectural definition of the instruction ... does not call for an interrupt" are discussed in section VI.A (page 100-101 of the specification), section VI.B (pages 101-103), many of the probeable events 610 in Fig. 4b, and many of the X86 transfer of control instructions discussed in section VI.D (pages 106-111).

This was apparently accepted as a sufficient explanation, because the issue was dropped from the Office Action of December 4, 2001.

7. "altering a manipulation of data or transfer of control behavior of the instruction in a manner incompatible with the architectural definition in an emulated architecture of the instruction"

The Office Action states as follows:

> The scope of the meaning of the following is not clear:
>
> ...
>
> 3. "altering a manipulation of data or transfer of control behavior of the instruction in a manner incompatible with the architectural definition in an emulated architecture of the instruction" of claim 39. The Examiner is unable to find the support or explanation of the clause.

a. No rejection is raised – the "rejected" language does not appear in claim 39

This language does not appear in claim 39. No rejection is raised.

b. This issue has been raised and resolved earlier in prosecution

This issue appears to have been raised in error. An almost identical issue was raised in the Office Action of April 10, 2001, and the following response was provided in the Response of August 10, 2001, at pages 15 and 17:

> **K. "alter—in manner incompatible with the architectural definition of the instruction"**
>
> The Examiner requested information as to the meaning of "alter — in manner incompatible with the architectural definition of the instruction" as used in claim 10.
>
> As is well known in the art, an architecture defines the behavior of each instruction in its instruction set. For example, most architectures define an "ADD" instruction to cause the addition of two numbers, a "JUMP" instruction to transfer control to another instruction in accordance with defined rules, and similar definitions for all other instructions in the instruction set. To "alter [an instruction's behavior] in a manner incompatible with the architectural definition of the instruction" is to cause an instruction to do something other than its architecturally-defined behavior. For example, an altered "ADD" instruction might perform a subtract, cause a transfer of control, or be an illegal instruction. An altered "JUMP" instruction might cause a control transfer to a destination other than the architecturally-defined destination, and/or cause a change in ISA under which further instructions are executed. Some examples are described in the specification at sections II (pages 44-46), IV (pages 64-73), VI.A (pages 100-101), VI.D (pages 106-111) and VI.E (pages 111-112). The specification may include other examples.
>
> Applicant believes that the plain language of the clause would be [well] understood by those in the art.

This was apparently accepted as a sufficient explanation, because the issue was dropped from the Office Action of December 4, 2001.

"Architectural definition of an instruction" is discussed in section IX.C.3.b at page 27 of this paper. "Data manipulation behavior" is explained at IX.C.4 at page 28 of this paper. "Transfer of control behavior" is discussed at section IX.C.5 at page 29.

Additional support in the specification may be found. One embodiment, using the probe bits 624 of PFAT 172 and I-TLB 116, is disclosed in Figs. 6a-6c and section VI (pages 100-116). A second embodiment, the CC bit 196, 200 of PFAT 172 and I-TLB 116, is disclosed in Figs.

G

2a-2c and section IV (pages 64-73). A third embodiment, the ISA bit 180, 182, 194 of PFAT 172 and I-TLB 116, is disclosed in Figs. 3a-3o and section II (pages 44-46).

8. "the architectural definition of the instruction with which the alteration is incompatible is a definition in an emulated architecture"

The Office Action states as follows:

> The scope of the meaning of the following is not clear:
>
> . . .
>
> 4. "the architectural definition of the instruction with which the alteration is incompatible is a definition in an emulated architecture" in claims 36-38.

"Architectural definition of an instruction" is discussed in section IX.C.3.b at page 27 of this paper. See section IX.C.7 at page 31 for a discussion of an "alteration" an instruction's behavior.

Nonetheless, purely to assist in examination of the application (and not in response to any statutory rejection – none exists), Applicant has amended certain claims from "architectural definition of an instruction in an emulated architecture" to the "architectural definition of an instruction in the instruction's native architecture." Either phrase uses only terms of art in their conventional sense. For example, when a RISC machine emulates an X86 ADD instruction, the phrase refers to the definition of the ADD instruction in the X86 architecture, as opposed to the definition of an ADD instruction in the RISC architecture.[7] Further, this amendment does not narrow the claims.

This phrase is composed of established terms of art, and thus no § 112 ¶ 2 rejection is warranted.

9. "logically equivalent"

The Office Action questions the use of the phrase "logically equivalent."

[7] It should be noted that this use of "native" is somewhat different than the use of "native" used in the specification, where "native" refers to the Tapestry RISC machine. However, because this use of "native" is always qualified by "the instruction's native architecture" or some similar phrase, the difference should always be clear in context.

a. This issue has been raised and resolved earlier in prosecution

This issue appears to have been raised in error. An almost identical issue was raised in the Office Action of April 10, 2001, and the following response was provided in the Response of August 10, 2001, at pages 19-20:

> **P. "logically equivalent"**
>
> The Examiner requested information as to the meaning of "equivalent" in claim 22....
>
> [A]t page 100, lines 10-17, the specification discusses one possibility for "logically equivalent" instruction text: RISC code, for example, produced by a binary translator, that performs similarly enough to the original X86 program that the two programs produce the same result.

This was apparently accepted as a sufficient explanation, because the issue was dropped from the Office Action of December 4, 2001.

10. "What actually the instruction pipeline circuitry does"

The Office Action queries, "With respect to claim 10, it is not clear what actually the instruction pipeline circuitry does."

a. This issue has been raised and resolved earlier in prosecution

This issue appears to have been raised in error. A similar issue was raised in the Office Action of April 2001. Applicant's paper of August 10, 2001 responded as follows, at pages 25-26:

> **E. "pipeline circuitry to effect control of instruction behavior"**
>
> In the Office Action, the Examiner requested information as follows:
>
> > Applicants are requested to identify the following components in the drawings and the description thereof in the specification: ...
> >
> > 5. the description of the pipeline circuitry to effect control of an architecturally-visible data manipulation behavior and control transfer behavior.
>
> The Examiner is referred to sections III.F [reproduced at section IX.C.4 at page 28 of this paper] and III.K [reproduced at section IX.C.7 at page 31 of this paper] of these Remarks. One specific embodiment is described in Figs. 6b and 6c, and discussed in section VI.D (pages 106-111) of the specification. Other examples are discussed in section VI.A (pages 100-101), and throughout section VI (pages 100-116).

This explanation was accepted in the Office Action of December 2001. Re-raising of a resolved issue appears to have been an oversight by the Examiner.

b. The claim itself recites “what actually the instruction pipeline circuitry does”

In pertinent part, claim 10 recites as follows (emphasis added):

> 10. A microprocessor chip, comprising:
> instruction pipeline circuitry;
>
> . . .
>
> the instruction pipeline circuitry being responsive to the contents of the table entry to alter a manipulation of data or control transfer behavior of the instruction in a manner incompatible with the architectural definition of the instruction in the instruction’s native architecture.

Claim 10 itself recites what the “instruction pipeline circuitry does:” it responds “to the contents of the table entry to alter a manipulation of data or control transfer behavior of the instruction in a manner incompatible with the architectural definition of the instruction in the instruction’s native architecture.” The instruction pipeline circuitry also performs the traditional functions of “instruction pipeline circuitry,” as that term is understood in the art.

In any particular “microprocessor chip,” it will be clear whether or not the chip includes “instruction pipeline circuitry” that responds to table contents as recited in the claim (in which case the embodiment meets the claim), or does not have such “instruction pipeline circuitry” (in which case it does not meet the claim). Section 112 ¶ 2 asks no more.

Unless it can be demonstrated that it might be ambiguous whether or not a particular computer does or does not include “instruction pipeline circuitry” as recited in the claim, a rejection under § 112 ¶ 2 is unwarranted.

c. An incorrect legal test has been applied

§ 112 ¶ 2 contains no requirement that a claim recite function for every structural element. Because an erroneous legal test has been applied, no rejection exists.

11. “Function of the lookup structure”

The Office Action queries, “Claim 14 fails to recite function of the lookup structure.” In pertinent part, Claim 14 recites as follows (emphasis added):

> 14. A microprocessor chip, comprising:
>
> . . .
>
> a lookup structure having entries … describing a likelihood of the existence of an alternate coding of instructions located in the respective corresponding address range.

The "function of the lookup structure" is to "describ[e] a likelihood of the existence of an alternate coding of instructions located in the … address range[s]" corresponding to the respective entries.

12. "Functional relationship between the circuits and the lookup structure such that meaningful operation can be achieved"

The Office Action queries, "Claim 14 further fails to recite functional relationship between the circuits and the lookup structure such that meaningful operation can be achieved."

a. Claim 14 itself recites a "functional relationship between the circuits and the lookup structure"

In pertinent part, claim 14 recites as follows (emphasis added):

> 14. A microprocessor chip, comprising:
> instruction pipeline circuitry;
>
> . . .
>
> a lookup structure having entries associated with corresponding address ranges generated by the instruction pipeline circuitry and translated by the address translation circuitry, …

The functional relationship is that the entries of the lookup structure are "associated with corresponding address ranges generated by the instruction pipeline circuitry and translated by the address translation circuitry."

b. No rejection can be raised on the stated grounds

"Functional relationship can be achieved" is, at best, a consideration under the "how to use" requirement of § 112 ¶ 1 or utility requirement under § 101. There is no such thing as a rejection half based on § 112 ¶ 1 and half based on § 112 ¶ 2 or § 101. MPEP § 2174 specifically forbids the "mix and match" approach to rejecting claims that is exemplified here:

> **2174 Relationship Between the Requirements of the First and Second Paragraphs of 35 U.S.C. 112**
>
> The requirements of the first and second paragraphs of 35 U.S.C. 112 are separate and distinct. …

Applicant respectfully requests that no future rejections be raised on grounds that are not authorized by the MPEP.

13. "First table and second table" of claim 57

The Office Action states as follows:

> Applicants are requested to identify the first table and the second table of claim 57 in the drawings and the description in the specification.

On its face, this is a request for information, not a rejection of any claim.

This question was answered in the Response of August 10, 2001, pages 10-11:

> The specification discusses several different "side tables" that are used to implement these inventions. … A second side table (PFAT 172, 174 with its probe bits 624) has entries that each correspond to regions of X86 instructions. The probe bits 624 of a single PFAT entry give an approximate indication, a likelihood estimate, of whether there is a translated code segment for any of the code in the region corresponding to the PFAT entry. A third side table (PIPM 602) is consulted when the PFAT table access indicates that existence of translated code is likely. The PIPM then gives the final definitive answer of whether execution should be transferred to RISC code, and where the relevant RISC code is located in memory.

Further discussion of the "first table" of claim 57 and its probe bits 624 of PFAT 172 and the I-TLB are discussed in section IX.C.14, below.

Embodiments of the "second table" of claim 57 include PIPM 602, shown in Figs. 1a and 6a-6c.

14. "Each entry describing a likelihood of the existence of an alternate coding of instructions"

The Office Action rejects a table whose "[entries describe] a likelihood of the existence of an alternate coding of instructions."

a. This issue has been raised and resolved earlier in prosecution

First, this issue appears to have been raised in error. The Response of February 4, 2002 noted as follows, at page 6:

> Section VI of the specification, “Statistical probing,” discusses the PFAT (page frame attribute table) 172. Entries of PFAT 172 correspond to memory pages. Each PFAT table entry includes five bits 624 (see Fig. 6b) that indicate, in an approximate, statistical way, the “likelihood of the existence of an alternate coding of instructions” on the corresponding page (*e.g.*, section VI.B (pages 101-103), section VI.C (pages 103-106), section VI.D (pages 106-111). PIPM 602 is used to resolve the uncertainty remaining after consulting the statistical information stored in bits 624 of PFAT 172.

This was clearly a satisfactory explanation, because the rejection over the Morley ‘982 reference was withdrawn based on it. This, in combination with the Examiner’s assurance in the interview of March 7, 2002 that all claim terms were sufficiently understood to examine the claims, suggests that a rejection for “indefiniteness” was not intended to be raised.

b. The Office Action fails to raise a legally-cognizable indefiniteness rejection

Second, the claims are drafted in reliance on MPEP § 2173.04, which specifically cautions that “Breadth Is Not Indefiniteness.” A claim need not recite a limitation when the claim is intended to cover all possibilities for that limitation. These claims are intended to cover both cases where the “alternate coding” exists, and cases where it does not, and the claims are drafted accordingly. The question asked in the Office Action is not properly raised in a § 112 ¶ 2 context.

Third, the Office Action does not raise any genuine issue of indefiniteness. For example, the Action does not suggest that one of ordinary skill would have any difficulty in determining whether or not a particular table does or does not meet the claims. This claim language is clear enough to determine that neither Morley ‘982 nor Adachi ‘975 (in combination with the other references cited in prior Office Actions) have such a table. Language that is understood is rarely indefinite.

Fourth, the Office Action appears to overlook the rule, repeated often throughout Chapter 2100 of the MPEP (emphasis added):

> **2111 Claim Interpretation; Broadest Reasonable Interpretation**
> **CLAIMS MUST BE GIVEN THEIR BROADEST REASONABLE INTERPRETATION**
> During patent examination, the pending claims must be “given the <u>broadest reasonable interpretation consistent with the specification</u>.” …

> **2164.08 Enablement Commensurate in Scope With the Claims**
> All questions of enablement are evaluated against the claimed subject matter. ... When analyzing the enabled scope of a claim, ... claims are to be given their broadest reasonable interpretation that is consistent with the specification.

In replacing the word "likelihood" with the word "probability," it appears that this requirement may have been neglected.

For these four reasons, no rejection exists.

c. The factual premises of any "rejection" are incorrect

Further, any rejection based on the language of "a likelihood of the existence of an alternate coding of instructions" is traversed for two further reasons.

Fifth, the statement that "a pipeline which is a digital circuit is able to respond to only precise instructions and not probability" (Office Action of March 2002, page 2, lines 12-14) is wrong. For example, most branch-prediction circuits store only indications of "likely" future program flow, not a "precise" prediction of future program flow. Analogously, in some implementations, the "likelihood of the existence" feature of the claims may provide an approximate early indication of future events or a likely approximation of a current state. In some implementations, that indication may be later confirmed or refuted with better information.

Sixth, in the interview of June 26, 2002, the Examiner indicated that he would prefer that this language be replaced with language more idiomatic to the art of branch prediction. Applicant submits that this claim language is idiomatic to the art, exemplified by claims 1 and 6 of U.S. Pat. No. 5,367,703, titled "Method and System for Enhanced Branch History Prediction Accuracy in a Superscalar Processor System," issued in 1994, assigned to I.B.M. These claims read as follows:

> 1. A method for enhanced branch history prediction accuracy in a superscalar processor system which is capable of fetching and dispatching up to N instructions simultaneously, said method comprising the steps of:
>
> establishing a branch history table containing multiple predictive fields, each of said multiple predictive fields containing data **indicative of a likelihood** that execution of a particular associated instruction will result in a branch within an executing set of instructions;
>
> ...

> utilizing an associated one of said M predictive fields to determine a **likelihood** that execution of a corresponding instruction within said ordered sequence of M instructions will …
>
> 6. A system …, said system comprising:
>
> means for establishing a branch history table containing multiple predictive fields, each of said multiple predictive fields containing data **indicative of a likelihood** that execution of a particular associated instruction will result in a branch within an executing set of instructions;
>
> …
>
> means for utilizing an associated one of said M predictive fields to **determine a likelihood** that execution of a corresponding instruction within said ordered sequence of M instructions will …

A number of other patents directed to branch prediction also use the word "likelihood" in a manner closely analogous to the usage in these claims. Note the variety of companies that have used this language – such broad use indicates that it is idiomatic to the art, and that this language is to "be read as [it] would be interpreted by those of ordinary skill in the art." MPEP § 2111.01.

patent	location	assignee
6,389,531	col. 26, lines 11-12	Hitachi, Ltd.
6,092,188	col. 3, lines 51-59	Intel Corp
6,065,115	col. 13, lines 53-60	Intel Corp
6,029,228	col. 24, line 55-58; col. 26, line 6	Texas Instruments Inc.
5,642,493	col. 4, line 27	Motorola, Inc.
4,860,197	col. 1, line 14	Prime Computer, Inc.
4,477,872	col. 1, line 34	I.B.M.

Further understanding probe bits 624 may be gained by considering the alternative embodiments shown in Figs. 6a, 6b and 6c, discussed in section VI.A (pages 100-101), at page 102, lines 6-19, and the first half of section VI.D at pages 106-108, and more generally throughout sections VI.B, VI.C and VI.D (pages 101-111). Probeable events 610 and the probe mask 620 are also relevant, as discussed sections VI.B, VI.C and VI.D (pages 101-111). The PFAT is shown in Figs. 1a, 1b and 1d, and discussed throughout the specification, particularly in section I.A (pages 29-31), section I.C (pages 35-36), section VI.B (pages 101-103), section VI.C (pages 103-106), section VI.D (pages 106-111), and section VI.G (pages 113-115).

D. Issues nominally arising under § 112 ¶ 1

The Office Action raises a number of "rejections" nominally based on § 112 ¶ 1.

1. The Office Action is insufficient to raise any enablement rejection of any claim

a. Almost all issues raised in the "enablement" section of the Office Action have been raised and previously resolved to the Examiner's satisfaction

Almost all of the enablement issues of this Office Action, questioning only where certain teaching exists in the specification, have been raised earlier in prosecution, in the guise of § 112 ¶ 2 "rejections." These previously-raised issues were resolved to the Examiner's satisfaction, as noted by the absence of these issues from the Office Action of December 2001. It is believed that the discussion in Applicant's previous papers fairly meets nearly all of the issues raised in the § 112 ¶ 1 portion of the March 2002 Office Action.

For each such resolved issue, the discussion previously found acceptable by the Examiner is repeated below.

b. The nominal § 112 ¶ 1 "rejections" apply the wrong legal test

Nearly all of the nominal § 112 ¶ 1 "rejections" raised in this Office Action give as the reason, "The specification fails to disclose." This is not a proper test under the "enablement" requirement of § 112 ¶ 1. The test for enablement is "undue experimentation." A claim limitation may well be "enabled" by the knowledge of one of ordinary skill, with no disclosure whatsoever in the specification.

MPEP § 2164.04 reads as follows (bold in original, quotations and citations omitted, underline added):

> **2164.04 Burden on the Examiner Under the Enablement Requirement**
>
> Before any analysis of enablement can occur, it is necessary for the examiner to construe the claims. For terms that are not well-known in the art, or for terms that could have more than one meaning, it is necessary that the examiner select the definition that he/she intends to use when examining the application, based on his/her understanding of what applicant intends it to mean, and explicitly set forth the meaning of the term and the scope of the claim when writing an Office action. ....
>
> In order to make a rejection, the examiner has the initial burden to establish a reasonable basis to question the enablement provided for the claimed invention. ...A specification disclosure which contains a teaching of the manner and process of making and using an invention in terms which correspond in scope to those used in describing and defining the subject matter sought to be patented must be taken as being in compliance with the enablement requirement of 35

> U.S.C. 112, first paragraph, unless there is a reason to doubt the objective truth of the statements contained therein which must be relied on for enabling support. … As stated by the court, "it is incumbent upon the Patent Office, whenever a rejection on this basis is made, to explain *why* it doubts the truth or accuracy of any statement in a supporting disclosure and to back up assertions of its own with acceptable evidence or reasoning which is inconsistent with the contested statement. Otherwise, there would be no need for the applicant to go to the trouble and expense of supporting his presumptively accurate disclosure."
>
> According to *In re Bowen*, 492 F.2d 859, 862-63, 181 USPQ 48, 51 (CCPA 1974), the minimal requirement is for the examiner to give reasons for the uncertainty of the enablement. …
>
> … For example, doubt may arise about enablement because information is missing about one or more essential parts or relationships between parts which one skilled in the art could not develop without undue experimentation. In such a case, the examiner should specifically identify what information is missing and why one skilled in the art could not supply the information without undue experimentation. …. However, specific technical reasons are **always** required.
>
> In accordance with the principles of compact prosecution, if an enablement rejection is appropriate, the first Office action on the merits should present the best case with all the relevant reasons, issues, and evidence so that all such rejections can be withdrawn if applicant provides appropriate convincing arguments and/or evidence in rebuttal. …

MPEP § 2164.04 states a number of requirements for an enablement rejection. In contrast, the Office Action merely accuses certain claim language, and meets none of the further analytical requirements set out in MPEP § 2164.04. Prosecution cannot advance when an Office Action fails to identify issues clearly, or state a rejection with sufficient detail to allow a focused response. An applicant should not be left to guess what the Examiner's unstated concerns might be.

c. Statements in the specification must be accepted at face value

Finally, the MPEP requires the Examiner to "take an applicant's word" for the utility and enablement of the specification (emphasis added):

> **2107.02 Procedural Considerations Related to Rejections for Lack of Utility**
> **An Asserted Utility Creates a Presumption of Utility**
>
> …
>
> As a matter of Patent Office practice, a specification which contains a disclosure of utility which corresponds in scope to the subject matter sought to be patented must be taken as sufficient to satisfy the utility requirement of § 101 for

> the entire claimed subject matter unless there is a reason for one skilled in the art to question the objective truth of the statement of utility or its scope.

See also the excerpts from § 2164.04 set out immediately above.

Applicant's papers have provided extensive indications of the locations in the specification that teach the how's and why's of the invention. Several more examples are presented throughout this paper. MPEP § 2164.05 warns that these statements in the specification must be presumed to be correct, and any enablement rejections must be supported by technical reasoning showing that the specification is incorrect: "The examiner should **never** make the [enablement] determination based on personal opinion" (bold and underline in original). Because there is no showing that any statement in the specification is incorrect or incredible, the Office Action is insufficient to raise any enablement rejection.

Nonetheless, even though no rejections have been raised, in order to effectively advance prosecution, the Examiner's concerns will be answered as best understood.

d. Rejection of language that does not appear in the claims

The Office Action purports to reject a number of phrases under § 112 ¶ 1. Even before amendment, most of these phrases did not appear in any claim. To consider one example, no claim recites "the interrupt criteria being based at least in part on the likelihood ..." Rather, claim 1, even before amendment, recited "the interrupt criteria [are] based at least in part on [a] table entry." Raising an interrupt based on a table entry would not require undue experimentation.

Because these paragraphs of the Office Action do not relate to any claim in the application, no rejection is raised.

2. "a table lookup circuitry having entries describing a likelihood of the existence of an alternate coding of instructions"

The Office Action states "The specification fails to disclose a table lookup circuitry having entries describing a likelihood of the existence of an alternate coding of instructions."

a. This "rejected" language does not appear in the claims; no "undue experimentation" has been shown

The "rejected" language does not appear in any claim, and the Office Action does not indicate which claim is intended. See further discussion of this problem at section IX.D.1.d at

G

page 43, above. Nor does the Office Action describe any basis to believe that "undue experimentation" would be required, or challenge the credibility of any statement in the specification. See further discussion of this problem at sections IX.D.1.b and IX.D.1.c at page 41, above. For these two reasons, no rejection is raised.

The closest language that does appear in the claims is discussed in section IX.C.14 at page 37, above.

b. This issue has been raised and resolved earlier in prosecution

This issue appears to have been raised in error. All of the elements of the phrase questioned by the Examiner have been discussed at some length during prior prosecution. See, for example, section IX.C.14 at page 37, above.

c. The Office Action is inadequate to raise a rejection

Because the location of supporting disclosure has been identified during prior prosecution, MPEP § 2164.04 requires that (bold and italic in original, underline added):

> **2164.04 Burden on the Examiner Under the Enablement Requirement**
>
> ...A specification disclosure which contains a teaching of the manner and process of making and using an invention in terms which correspond in scope to those used in describing and defining the subject matter sought to be patented must be taken as being in compliance with the enablement requirement of 35 U.S.C. 112, first paragraph, unless there is a reason to doubt the objective truth of the statements contained therein which must be relied on for enabling support. ... As stated by the court, "it is incumbent upon the Patent Office, whenever a rejection on this basis is made, to explain *why* it doubts the truth or accuracy of any statement in a supporting disclosure and to back up assertions of its own with acceptable evidence or reasoning which is inconsistent with the contested statement.
>
> According to *In re Bowen*, 492 F.2d 859, 862-63, 181 USPQ 48, 51 (CCPA 1974), the minimal requirement is for the examiner to give reasons for the uncertainty of the enablement. ...
>
> ... For example, doubt may arise about enablement because information is missing about one or more essential parts or relationships between parts which one skilled in the art could not develop without undue experimentation. In such a case, the examiner should specifically identify what information is missing and why one skilled in the art could not supply the information without undue experimentation. .... However, specific technical reasons are **always** required.

A mere indication of claim language and a bald statement that "The specification does not disclose…" is inconsistent with earlier actions taken by the Examiner, and fails to meet the requirement for "specific technical reasons" for believing undue experimentation might be required.

Because steps required to raise a rejection have not been performed, no rejection exists.

3. "interrupt circuitry which triggers an interrupt in accordance with interrupt criteria on execution of an instruction, wherein the architectural definition of the instruction does not call for an interrupt, the interrupt criteria being based at least in part on the likelihood of the existence of an alternate coding of instructions (probability)"

The Office Action states "The specification fails to disclose an interrupt circuitry which triggers an interrupt in accordance with interrupt criteria on execution of an instruction, wherein the architectural definition of the instruction does not call for an interrupt, the interrupt criteria being based at least in part on the likelihood of the existence of an alternate coding of instructions (probability)."

a. This "rejected" language does not appear in the claims; no "undue experimentation" has been shown

The "rejected" language does not appear in any claim, and the Office Action does not indicate which claim is intended. See further discussion of this problem at section IX.D.1.d at page 43, above. Nor does the Office Action describe any basis to believe that "undue experimentation" would be required, or challenge the credibility of any statement in the specification. See further discussion of this problem at sections IX.D.1.b and IX.D.1.c at page 41, above. For these two reasons, no rejection is raised.

b. This issue has been raised and resolved earlier in prosecution

This issue appears to have been raised in error. Nearly identical issues have been raised and resolved to the Examiner's satisfaction earlier in prosecution. For example, the Response of August 10, 2001 reads as follows, at page 21:

> **C. "interrupt circuitry and the handler"**
>
> In the Office Action, the Examiner requested information as follows:

> Applicants are requested to identify the following components in the drawings and the description thereof in the specification: ...
>
> 3. the interrupt circuitry and the handler,
>
> One particular example of the interrupt circuitry and handler are disclosed in Figs. 6b and 6c, and discussed in section VI.D (pages 106-111) of the specification.[8] Other examples are discussed throughout section VI (pages 100-116). Other examples include the hardware and software that handle the ISA flag bit 180 (section II, pages 44-46), and the CC flag bit 200 (section IV, pages 64-73).

This explanation was deemed fully acceptable by the Examiner, as evidenced by the withdrawal of this concern in the Office Action of December 2001.[9]

The next component of the phrase queried about in the current Office Action, "architectural definition of the instruction in an emulated architecture does not call for an interrupt," was similarly addressed in the same Response at page 14, as discussed in section IX.C.6 at page 30 of this paper. That explanation was also deemed acceptable, because the question was not re-raised in the Action of December 2001. To applicant's knowledge, this element is not known in the art, but it is enabled and could be practiced without undue experimentation, as discussed in the portions of the specification indicated in section IX.C.6.

The remainder of this phrase, referring to the "likelihood," is discussed in section IX.C.14 at page 37 of this paper. As noted there, this phrase also was fully explained to the Examiner's satisfaction in Applicant's Response of August 10, 2001.

For reasons discussed in section IX.D.2 at page 44, no rejection exists.

4. "a handler being responsive to the likelihood of the existence of an alternate coding of instructions to affect the instruction pipeline circuitry to effect control of an architecturally-visible data manipulation behavior or control transfer behavior of the instruction"

The Office Action states, "The specification fails to disclose a handler being responsive to the likelihood of the existence of an alternate coding of instructions to affect the instruction

[8] To narrow this indication somewhat, a first embodiment of the interrupt circuitry is shown in Fig. 6b, the upper half of Fig. 6c, and the first half of section VI.D at pages 106-108. Section VI (pages 101-111) is relevant in its entirety.

[9] Other examples of an "interrupt handler being responsive to a table entry" are shown in the lower half of Fig. 6c, and at section VI.D, especially pages 109-110.

pipeline circuitry to effect control of an architecturally-visible data manipulation behavior or control transfer behavior of the instruction as claimed."

a. This "rejected" language does not appear in the claims; no "undue experimentation" has been shown

The "rejected" language does not appear in any claim, and the Office Action does not indicate which claim is intended. See further discussion of this problem at section IX.D.1.d at page 43, above. Nor does the Office Action describe any basis to believe that "undue experimentation" would be required, or challenge the credibility of any statement in the specification. See further discussion of this problem at sections IX.D.1.b and IX.D.1.c at page 41, above. For these two reasons, no rejection is raised.

b. This issue has been raised and resolved earlier in prosecution

This issue appears to have been raised in error. Nearly identical issues have been raised and resolved to the Examiner's satisfaction earlier in prosecution.

For example, the "handler" was discussed in the Response of August 10, 2001, and that explanation was accepted in the Action of December 2001, as explained in section IX.D.3 at page 45 of this paper.

The "table whose entries indicate a likelihood of the existence of an alternate coding" was discussed in the Response of August 10, 2001, and that explanation was accepted in the Action of December 2001, as explained in section IX.C.14 at page 37 of this paper.

"Effecting control of an architecturally-visible data manipulation behavior or control transfer behavior" was explained to the Examiner's satisfaction in the Response of August 10, 2001, as discussed in sections IX.C.4 and IX.C.5, at pages 28 and 29 of this paper.

For reasons discussed in section IX.D.2 at page 44, no rejection exists.

5. "an instruction pipeline circuitry being affected by the handler being responsive to the likelihood of the existence of an alternate coding of instructions to effect control of an architecturally-visible data manipulation behavior or control transfer behavior of the instruction"

The Office Action states, "The specification fails to disclose an instruction pipeline circuitry being affected by the handler being responsive to the likelihood of the existence of an alternate coding of instructions to effect control of an architecturally-visible data manipulation behavior or control transfer behavior of the instruction as claimed."

a. This "rejected" language does not appear in the claims; no "undue experimentation" has been shown

The "rejected" language does not appear in any claim, and the Office Action does not indicate which claim is intended. See further discussion of this problem at section IX.D.1.d at page 43, above. Nor does the Office Action describe any basis to believe that "undue experimentation" would be required, or challenge the credibility of any statement in the specification. See further discussion of this problem at sections IX.D.1.b and IX.D.1.c at page 41, above. For these two reasons, no rejection is raised.

b. This issue has been raised and resolved earlier in prosecution

This issue appears to have been raised in error. Nearly identical issues have been raised and resolved to the Examiner's satisfaction earlier in prosecution. For example, the Response of August 10, 2001 provides a description of "instruction pipeline circuitry being affected by the handler," at page 19:

> **O. "Function of the interrupt handler"**
>
> The Examiner requested information as to the meaning of the "Function of the interrupt handler" as recited in claim 21. Claims 20 and 21 recite, in pertinent part:
>
> > 20. The microprocessor chip of claim 19, further comprising:
> >
> > interrupt handler software designed to service the interrupt and to return control to an instruction flow of the process other than the instruction flow triggering the interrupt, the returned-to instruction flow for carrying on non-error handling normal processing of the process.
> >
> > 21. The microprocessor chip of claim 20, wherein the interrupt handler software is programmed to change an instruction set architecture under which instructions are interpreted by the computer.
>
> As the Examiner observes, the interrupt handler of claim 21 does some things that are not "common." Claim 20 recites that the handler is "designed to service the interrupt and to return control to an instruction flow of the process other than the instruction flow triggering the interrupt." Claim 21 recites that "the interrupt handler software is programmed to change an instruction set architecture."

This explanation was regarded as sufficient, because the issue was dropped from the Action of December 2001.

Further examples of "instruction pipeline circuitry being affected by the handler" are shown in lower half of Fig. 6c; section VI.D, especially pages 109-110; see also section VI (pages 100-116), section II (pages 44-46), and section IV (pages 64-73).

Any issue relating to the language "being responsive to the likelihood of the existence of an alternate coding of instructions" has been resolved earlier in prosecution, and re-raising a rejection now is inappropriate, as discussed in section IX.C.14 at page 37 above.

"Effecting control of an architecturally-visible data manipulation behavior or control transfer behavior" was explained to the Examiner's satisfaction in the Response of August 10, 2001, as discussed in sections IX.C.4 and IX.C.5, at pages 28 and 29 of this paper.

For reasons discussed in section IX.D.2 at page 44, no rejection exists.

6. "an instruction pipeline circuitry being responsive to the likelihood of the existence of an alternate coding of instructions to alter a manipulation behavior or control transfer behavior of the instruction in a manner incompatible with the architectural definition of the instruction"

The Office Action states "The specification fails to disclose an instruction pipeline circuitry being responsive to the likelihood of the existence of an alternate coding of instructions to alter a manipulation behavior or control transfer behavior of the instruction in a manner incompatible with the architectural definition of the instruction as claimed in claims 10 and 39 for example."

a. This "rejected" language does not appear in the claims; no "undue experimentation" has been shown

The "rejected" language does not appear in either claim 10 or 39. See further discussion of this problem at section IX.D.1.d at page 43, above. Nor does the Office Action describe any basis to believe that "undue experimentation" would be required, or challenge the credibility of any statement in the specification. See further discussion of this problem at sections IX.D.1.b and IX.D.1.c at page 41, above. For these two reasons, no rejection is raised.

b. This issue has been raised and resolved earlier in prosecution

This issue appears to have been raised in error. Any issue relating to the language "being responsive to the likelihood of the existence of an alternate coding of instructions" has been resolved earlier in prosecution, as discussed in section IX.C.14 at page 37 above. Re-raising a rejection now is inappropriate.

"Altering a data manipulation behavior or control transfer behavior of the instruction" was explained to the Examiner's satisfaction in the Response of August 10, 2001, as discussed in sections IX.C.4 and IX.C.5, at pages 28 and 29 of this paper.

"Altering an [instruction's behavior] in a manner incompatible with the architectural definition of the instruction" is a resolved issue, as explained in section IX.C.7 at page 31.

For reasons discussed in section IX.D.2 at page 44, no rejection exists.

7. "control of architecturally-visible data manipulation behavior includes changing an instruction set architecture under which instructions are interpreted"

The Office Action states "The specification fails to disclose the control of architecturally-visible data manipulation behavior includes changing an instruction set architecture under which instructions are interpreted by the computer of claim 5."

The Office Action does not describe any basis to believe that "undue experimentation" would be required, nor challenge the credibility of any statement in the specification. Accordingly, no rejection exists.

Examples are found throughout the specification. Attention is drawn specifically to section II (pages 44-46), page 110, lines 6-8; see generally section VI (pages 100-116).

8. "an interrupt circuitry to trigger an interrupt in accordance with synchronous interrupt criteria being based on a memory state and wherein the architectural definition of the instruction in an emulated architecture does not call for an interrupt"

The Office Action states "The specification fails to disclose an interrupt circuitry to trigger an interrupt in accordance with **synchronous** interrupt criteria being based on a **memory state** and wherein the architectural definition of the instruction in **an emulated architecture** does not call for an interrupt" (emphasis in original).

"Synchronous interrupt" is an established term of art, as discussed above in section IX.D.9 at page 51. No undue experimentation is required.

G

"Memory state" is an established term of art, as discussed above in section IX.D.10 at page 52. "Synchronous interrupts" based on "a memory state" do not require undue experimentation – for example, TLB fills and page faults are known.

"Triggering an interrupt [for an instruction whose] architectural definition ... does not call for an interrupt" is discussed in section IX.C.6 at 30.

"Architectural definition of an instruction" is discussed in section IX.C.3.b at page 27 of this paper.

One embodiment is described particularly in Fig. 6b, and the upper half of Fig. 6c. Also relevant are col. 610 of Fig. 4b, page 106, lines 9-20, see also section VI.A (page 100-101), section VI.B (pages 101-103), and section VI.D (pages 106-111).

9. "Synchronous interrupt criteria"

The Office Action states, "The specification ... fails to explain what synchronous interrupt criteria .. are."

The Examiner may recall that claim 1, as originally filed, recited

> interrupt circuitry cooperatively designed with the instruction pipeline circuitry to trigger an interrupt on execution of an instruction of a process, synchronously based at least in part on a memory state of the computer and the address of the instruction...

The Office Action of April 2001 rejected claim 1 because "Claim 1 fails to recite what event causes the interrupt." Even though the rejection was factually and legally groundless[10], the claim was amended to explicitly recite "criteria" that cause the interrupt. In this Response, the claim is amended largely to its original form, to eliminate the word "criteria" and return to the original and unarguably well-established term, "synchronous interrupt."

Applicant does not concede that any enablement rejection was properly raised – MPEP § 2164.01 instructs that "A patent need not teach, and preferably omits, what is well known in the art." "Synchronous interrupts" are known in the art, and no "undue experimentation" would be required to design a computer that raised interrupts based on desired synchronous interrupt

[10] The claim recited that the interrupt was triggered "based at least in part on a memory state of the computer and the address of the instruction." The statement that "fails to recite what event causes the interrupt" was simply incorrect. Further, there is no requirement under § 112 ¶ 2 that a prior cause be recited for every step in a claim.

G

criteria. A quick search of the U.S. patent database on Westlaw reveals that seventy patents issued since 1970 use the phrase "synchronous interrupt." See also Tanenbaum, Exhibit 3, pages 263-269; Hennessey & Patterson, Exhibit 4, pages 214-220; Intel manual, vol. 1, Exhibit 5, pages 4-10 to 4-11. No enablement rejection is properly raised.

Synchronous interrupts arise out of the execution of instructions. Examples include TRAP or SVC exceptions, divide-by-zero exceptions, page faults, and TLB misses. In contrast, "asynchronous interrupts" are not directly connected to an instruction. Examples of asynchronous interrupts include most interrupts from peripheral devices, timers, or power-fail.

The claim language "synchronous interrupt," or analogous concepts, are supported at page 18, line 29; page 19, line 16; page 25, line 29; claim 1 (as originally filed), line 12; Fig. 6b, the upper half of Fig. 6c; by section VI of the specification (pages 101-117), particularly the discussion of the probe interrupt discussed in and the first half of section VI.D at pages 106-108; by the ISA-change exception described in sections II (pages 44-46), III.C-III.H (pages 51-64); or by the calling-convention change exception described in section IV (pages 64-73); see also Figs. 3a-3o and section II (pages 44-46); Figs. 2a-2c and section IV (pages 64-73).

10. "Memory state"

"Memory state" is an established term of art. For example, a Westlaw search of the patent database reveals that over 2,400 patents have used the term "memory state" or "state of memory" since 1970. For example, the second sentence of U.S. Pat. No. 6,035,376 reads: "The present invention relates to a system and method for maintaining cache coherence that is event driven and changes the state of the caches and memories based on the current memory state and a head of a list of corresponding cache entries." U.S. Pat. No. 5,829,032 uses the sentence "A memory tag state controller 126 generates a state signal for cache coherency based on the memory tag state." One of ordinary skill in the art would not have to resort to undue experimentation to achieve a desired memory state, or to have hardware respond as desired to a memory state.

Various recitations of "memory state" are supported by one or more of the following: Fig. 1a; Figs. 2a-2c; page 18, line 19; page 19, line 17; claim 1 (as originally filed), line 13; section II (pages 44-46), particularly the discussion of ISA bits 180, 182, 194; section IV (pages 64-73), particularly the discussion of the "calling convention" bit 196, 200; Figs. 6a-6c, and section VI of the specification (pages 110-116), particularly the discussion of probe bits 624 and the PIPM table 602 (Physical IP Map, described in most detail in section VI.A and VI.D, pages 110-112 and 108-111), and page 102, lines 6-19.

11. what the wherein clause ["wherein the architectural definition of the instruction in an emulated architecture does not call for an interrupt"] means

The Office Action states "The Examiner is unable to find the explanation of the wherein clause in the specification."

Earlier in prosecution, the Examiner requested the same information, and it was provided to the Examiner's satisfaction. See section IX.C.6 at page 30.

If there is to be an enablement rejection, the burden is on the Examiner to show a reasonable basis to believe that some statement is unlikely to be true. See MPEP §§ 2164.04 and 2164.05, and sections IX.D.1.b and IX.D.1.c at page 41. Because the Office Action attempts no such showing, no rejection exists.

In view of the amendments and remarks, Applicant respectfully submits that no claim is rejected by the Office Action of March 2002. Further, the claims are in condition for allowance. Applicant requests that the application be passed to issue in due course. The Examiner is urged to telephone Applicant's undersigned counsel at the number noted below if it will advance the prosecution of this application, or with any suggestion to resolve any condition that would impede allowance. In the event that any further extension of time is required, Applicant petitions for that extension of time required to make this response timely. Kindly charge any additional fee, or credit any surplus, to Deposit Account 50-0675, Order No. 5231.16-4004C.

Respectfully submitted,

SCHULTE ROTH & ZABEL

Dated: July 8, 2002

By: David E. Boundy
David E. Boundy
Registration No. 36,461

Mailing Address:

G

SCHULTE ROTH & ZABEL
919 Third Avenue
New York, New York 10022
(212) 756-2000
(212) 593-5955 Telecopier

G

EXHIBIT 2

VERSION OF REWRITTEN CLAIMS MARKED UP TO SHOW CHANGES

1. (three times amended, last amended 7/8/2002) A microprocessor chip, comprising:

instruction pipeline circuitry; and

table lookup circuitry designed to retrieve an entry from a table, each entry of the table being associated with a corresponding address range of an address space translated by address translation circuitry of the microprocessor chip, each entry describing a likelihood of the existence of an alternate coding of instructions located in the respective corresponding address range, the table lookup circuitry operable as part of the basic instruction cycle of executing an instruction of a non-supervisor mode program for execution on the microprocessor chip [executing on a computer], the table being stored in storage that is architecturally invisible to programs in the native architecture of at least some instructions executed by the microprocessor chip;

interrupt circuitry cooperatively designed with the instruction pipeline circuitry to [synchronously] trigger a synchronous [an] interrupt [in accordance with interrupt criteria] on execution of an instruction of a process, wherein the architectural definition of the instruction of the process does not call for an interrupt, a trigger for the interrupt [criteria] being synchronously based at least in part on the table entry corresponding to [associated with] the address of the instruction of the process, the interrupt circuitry being designed to invoke a handler for the interrupt, the handler being responsive to a content of the table entry to affect the instruction pipeline circuitry to effect control of an architecturally-visible data manipulation behavior or control transfer behavior of the instruction of the process, based at least in part on the contents of a table entry corresponding to [associated with] the address range in which the instruction of the process lies.

2. (three times amended, last amended 7/8/2002) A method, comprising the steps of:

as part of the basic instruction cycle of executing an instruction of a non-supervisor mode program executing on a computer, consulting a table, the table having entries that are [being] indexed by the address within an address space of instructions executed, entries of the table containing attributes of instructions whose addresses index to the respective table entries; and

controlling an architecturally-visible data manipulation behavior or control transfer behavior of the instruction based at least in part on a content of a table entry indexed by [associated with] the address of the instruction.

3. (twice amended, last amended 7/8/2002) The method of claim 2, wherein the control of control transfer behavior includes transfer of execution control to a second instruction for execution, the second instruction being an instruction other than an instruction architecturally-defined to be the successor instruction of the instruction.

9. (twice amended, last amended 7/8/2002) The method of claim 2, further comprising the steps of:

[synchronously] triggering a synchronous [an] interrupt on execution of the [an] instruction [of a process in accordance with interrupt criteria, the interrupt criteria being] based at least in part on a memory state of the computer and the address of the instruction, wherein the architectural definition of the instruction in the instruction's native [an emulated] architecture does not call for an interrupt.

10. (twice amended, last amended 7/8/2002) A microprocessor chip, comprising:

instruction pipeline circuitry;

table lookup circuitry designed to index into a table by a memory address of a memory reference arising during execution of an architecturally-defined instruction, and to retrieve a table entry corresponding to the address, the table entry being distinct from the memory referenced by the memory reference;

the instruction pipeline circuitry being responsive to the contents of the table entry to alter a manipulation of data or control transfer [of control] behavior of the instruction in a manner incompatible with the architectural definition of the instruction in the instruction's native architecture.

11. (three times amended, last amended 7/8/2001) The microprocessor chip of claim 10, further comprising:

a binary translator programmed to translate at least a selected portion of a computer program from a first binary representation to a second binary representation; and

wherein the pipeline control circuitry is further designed to initiate a determination of whether to transfer control from an execution of the architecturally-defined instruction, the architecturally-defined instruction being an instruction of the first binary representation of the program, to the second binary representation, and effective to initiate the determination with neither a query nor a control transfer [of control] to the second binary representation being coded into the first binary representation.

12. (twice amended, last amended 7/8/2002) The microprocessor chip of claim 10, further comprising:

interrupt circuitry cooperatively designed with the instruction pipeline circuitry to trigger a synchronous [an] interrupt on execution of the architecturally-defined instruction [in accordance with interrupt criteria, the interrupt criteria being] based at least in part on the contents of the table entry [a memory state of the computer and the address of the instruction], wherein the architectural definition of the architecturally-defined instruction in the instruction's native [an emulated] architecture does not call for an interrupt.

13. (twice amended, last amended 7/8/2001) The microprocessor chip of claim 12, further comprising:

interrupt handler software designed to service the interrupt and to effect the altering of a control transfer [of control] behavior, the altering being in the form of returning control from the handler to an instruction flow [of the process] other than the instruction flow triggering the interrupt, the returned-to instruction flow for carrying on non-error handling normal processing logically equivalent to the architecturally-defined processing of instructions following the altered instruction [of the process].

17. (amended, last amended 7/8/2002) The microprocessor chip of claim 14, wherein:

the instruction pipeline circuitry is responsive to the contents of the lookup structure entry to affect an architecturally-visible manipulation of data or control transfer [of control] behavior of [defined for] the instruction.

18. (twice amended, last amended 7/8/2002) The microprocessor chip of claim 14, further comprising:

interrupt circuitry cooperatively designed with the instruction pipeline circuitry to trigger a synchronous [an] interrupt on execution of an instruction of a process [, synchronously based on interrupt criteria, the interrupt criteria] based at least in part on a lookup structure entry associated with [a memory state of the computer and] the address of the instruction, wherein the architectural definition of the instruction in the instruction's native [an emulated] architecture does not call for an interrupt.

19. (four times amended, last amended 7/8/2002) A microprocessor chip, comprising:

instruction pipeline circuitry; and

interrupt circuitry cooperatively designed with the instruction pipeline circuitry to trigger a synchronous [an] interrupt on execution of an instruction of a process [in accordance with synchronous interrupt criteria, the interrupt criteria being] based at least in part on a memory state of the computer and the address of the instruction, wherein the architectural definition of the instruction in the instruction's native [an emulated] architecture does not call for an interrupt.

23. (amended, last amended 7/8/2002) The microprocessor chip of claim 19, further comprising:

table lookup circuitry designed to index into a table by a memory address within an address space of a memory reference arising during execution of the [an] instruction, and to retrieve a table entry corresponding to the memory-reference address;

the instruction pipeline circuitry being responsive to the contents of the table entry to affect an architecturally-visible manipulation of data or control transfer [of control] behavior of [defined for] the instruction.

26. (twice amended, last amended 7/8/2002) The method of claim 24, further comprising the step of:

altering a behavior of the instruction in a manner incompatible with the architectural definition [in an emulated architecture] of the instruction in the instruction's native architecture, based at least in part on a content of the lookup structure entry corresponding to the address range containing the instruction.

27. (twice amended, last amended 7/8/2001) The method of claim 24,

wherein each lookup structure entry corresponds to a page managed by a virtual memory manager, and wherein circuitry for locating a lookup structure [an] entry [of the table] is integrated with virtual memory address translation circuitry of the computer.

28. (twice amended, last amended 7/8/2002) The method of claim 24, further comprising the step of:

based at least in part on a content of the lookup structure entry, transferring control to the alternative coding, the alternative coding being an instruction flow of the process other than the instruction flow triggering the consulting, the transferred [returned]-to instruction flow being programmed to carry on non-error handling normal processing of the process.

29. (twice amended, last amended 7/8/2002) The method of claim 24, further comprising the step of:

based at least in part on a content of the lookup structure entry, [synchronously] triggering a synchronous [an] interrupt [in accordance with interrupt criteria, the interrupt criteria being based at least in part on a memory state of the computer and the address of the instruction], wherein the architectural definition of the instruction in the instruction's native [an emulated] architecture [of the instruction] does not call for an interrupt.

G

30. (twice amended, last amended 7/8/2001) A method, comprising the steps of:

on execution of an instruction of a process in a computer, [synchronously] triggering a synchronous [an] interrupt based at least in part on a memory state of the computer and the address of the instruction, wherein the architectural definition of the instruction in the instruction's native architecture does not call for an interrupt.

32. (amended 7/8/2002) The method of claim 31, further comprising the step of:

changing an instruction set architecture under which instructions are interpreted by the computer in [the] handler software for the interrupt.

34. (twice amended, last amended 7/8/2002) The method [microprocessor chip] of claim 30, further comprising:

[table lookup circuitry designed to] indexing into a table by a memory address within an address space of a memory reference arising during execution of an instruction, and to retrieve a table entry corresponding to the address;

responding [the instruction pipeline circuitry being responsive] to a content of the table entry to affect an architecturally-visible manipulation of data or control transfer [of control] behavior of [defined for] the instruction.

35. (added 1/9/2001) The method of claim 2, wherein the table entry indexed by the address of the instruction is associated with a range of instruction addresses.

Kindly cancel claims 36, 37 and 38 without prejudice or disclaimer.

39. (amended, last amended 7/8/2002) A method, comprising the steps of:

as part of the basic instruction cycle of executing an architecturally-defined instruction of a non-supervisor mode program executing on a computer, retrieving an entry from a table, the [entry of the] table entry being indexed by the address of a memory reference arising during execution of the architecturally-defined instruction, the table entry being distinct from the memory referenced by the memory reference;

based at least in part on a content of the table entry, altering a manipulation of data or control transfer [of control] behavior of the architecturally-defined instruction in a manner incompatible with the architectural definition [in an emulated architecture] of the architecturally-defined instruction in the architecturally-defined instruction's native architecture.

41. (amended, last amended 7/8/2002) The method of claim 39, wherein the [entry of the] table entry is indexed by the address within an address space of instructions fetched for execution [executed].

42. (amended, last amended 7/8/2002) The method of claim 39, wherein:

entries of the table correspond to respective address ranges, and the table entries describe a likelihood of the existence of an alternate coding of instructions located in the respective corresponding address ranges.

43. (amended, last amended 7/8/2002) The method of claim 39:

wherein the architectural definition of the architecturally-defined instruction in the architecturally-defined instruction's native [emulated] architecture does not call for an interrupt;

and further comprising the step of [synchronously] triggering a synchronous [an] interrupt based at least in part on a memory state of the computer and the address of the architecturally-defined instruction.

44. (amended, last amended 7/8/2002) The method of claim 39, wherein the control of control transfer [of control] behavior includes transfer of execution control to a second instruction for execution.

45. (amended, last amended 7/8/2002) The method of claim 44, wherein the second instruction is coded in an instruction set architecture (ISA) different than the ISA of the architecturally-defined [executed] instruction.

Kindly cancel claim 49.

50. (amended, last amended 7/8/2002) An apparatus, comprising:

instruction pipeline circuitry; and

table lookup circuitry designed to retrieve a table entry from a table whose entries are indexed by an address within an address space of an instruction fetched for execution by the instruction pipeline circuitry;

the instruction pipeline circuitry being responsive to a content of the table entry to control an architecturally-visible data manipulation behavior or control transfer behavior of the fetched instruction based at least in part on a content of the table entry indexed by [associated with] the address of the instruction.

51. (amended, last amended 7/8/2002) The apparatus [method] of claim 50, wherein the control of control transfer behavior includes transfer of execution control to a second instruction for execution, the second instruction being coded in an instruction set architecture (ISA) different than the ISA of the executed instruction.

52. (amended, last amended 7/8/2002) The apparatus [method] of claim 50,

wherein entries of the table correspond to pages managed by a virtual memory manager, circuitry for locating a table entry being integrated with virtual memory address translation circuitry of the computer.

Kindly add the following new claim 53:

53. (new, added 7/8/2002) The apparatus of claim 50, wherein:

the table entry is stored in storage architecturally invisible to programs executing in the instruction's native architecture.

Kindly amend claims 54-58 as follows:

54. (amended, last amended 7/8/2002) The apparatus [method] of claim 50, further comprising [the steps of]:

circuitry designed to trigger a synchronous [triggering an] interrupt on execution of an instruction of a process, [synchronously] based at least in part on a memory state of the computer and the address of the instruction, wherein [when] the architectural definition of the instruction in the instruction's native [an emulated] architecture does not call for an interrupt.

55. (amended, last amended 7/8/2002) The apparatus [method] of claim 50, wherein: the table entries are indexed by a virtual address of the instruction.

56. (amended, last amended 7/8/2002) The apparatus [method] of claim 50, wherein: the table entries are indexed by a physical address of the instruction.

58. (amended 7/8/2002) The microprocessor chip of claim 19, wherein a trigger for the interrupt is [criteria are] further based on a [the] value of the instruction.

Kindly add the following new claims.

59. (new 7/8/2002) The microprocessor chip of claim 19, wherein:

the memory state on which triggering the interrupt is based includes an entry of a table indexed by the address of instructions fetched for execution, entries of the table containing attribute indicia of instructions whose addresses index to the respective entries, the table entries being architecturally invisible to an architecture for execution in the instruction pipeline circuitry.

60. (new 7/8/2002) The method of claim 24, wherein:

the lookup structure entries are architecturally invisible to programs executing in an instruction set architecture executed by the instruction pipeline circuitry.

G

61. (new 7/8/2002) A method, comprising the steps of:

as part of the basic instruction cycle of executing an instruction of a non-supervisor mode program fetched for execution on a computer, consulting a table, entries of the table being indexed by addresses of instructions fetched, entries of the table containing attribute indicia of instructions whose addresses index to the respective entries; and

controlling an architecturally-visible data manipulation behavior or control transfer behavior of the fetched instruction based at least in part on a content of a table entry indexed by the address of the fetched instruction, the table entries being architecturally-invisible in the fetched instruction's native architecture.

62. (new 7/8/2002) The method of claim 61, wherein the control of control transfer behavior includes transfer of execution control to a second instruction for execution, the second instruction being an instruction other than an instruction architecturally-defined to be the successor instruction of the fetched instruction.

63. (new 7/8/2002) The method of claim 62, wherein the second instruction is coded in an instruction set architecture (ISA) different than the ISA of the fetched instruction.

64. (new 7/8/2002) The method of claim 61, wherein the control of architecturally-visible data manipulation behavior includes changing an instruction set architecture under which instructions are interpreted by the computer.

65. (new 7/8/2002) The method of claim 61, wherein the behavior control includes selecting between two different instruction set architectures, and the computer includes instruction pipeline circuitry designed to effect interpretation of computer instructions under the two instruction set architectures alternately.

G

66. (new 7/8/2002) The method of claim 61, wherein:

the attribute indicia describe a likelihood of the existence of an alternate coding of instructions located in respective address ranges corresponding to the table entries.

67. (new 7/8/2002) The method of claim 61:

wherein the architectural definition of the fetched instruction in the fetched instruction's native architecture does not call for an interrupt;

and further comprising the step of triggering a synchronous interrupt based at least in part on a memory state of the computer and the address of the fetched instruction.

68. (new 7/8/2002) The method of claim 61, further comprising the step of:

altering a behavior of the fetched instruction in a manner incompatible with the architectural definition of the fetched instruction in the fetched instruction's native architecture, based at least in part on a content of the table entry indexed by the address of the fetched instruction.

69. (new 7/8/2002) The method of claim 61, wherein:

entries of the table correspond to pages managed by a virtual memory manager, and wherein circuitry for locating an entry of the table is integrated with virtual memory address translation circuitry of the computer.

70. (new 7/8/2002) An apparatus, comprising:

instruction pipeline circuitry; and

table lookup circuitry designed to retrieve a table entry from a table whose entries are indexed by an address of an instruction fetched for execution, the table being stored in storage that is architecturally invisible to programs in the fetched instruction's native architecture;

the instruction pipeline circuitry being responsive to a content of the table entry to control an architecturally-visible data manipulation behavior or control transfer behavior of

G

the fetched instruction based at least in part on a content of the table entry associated with the address of the fetched instruction.

71. (new 7/8/2002) The apparatus of claim 70, wherein the control of control transfer behavior includes transfer of execution control to a second instruction for execution, the second instruction being coded in an instruction set architecture (ISA) different than the ISA of the fetched instruction.

72. (new 7/8/2002) The apparatus of claim 70,

wherein entries of the table correspond to pages managed by a virtual memory manager, the table lookup circuitry being integrated with virtual memory address translation circuitry.

73. (new 7/8/2002) The apparatus of claim 70, wherein:

entries of the table describe a likelihood of the existence of an alternate coding of instructions located in address ranges corresponding to respective table entries.

74. (new 7/8/2002) The apparatus of claim 70, wherein:

the control of the fetched instruction's behavior includes altering a manipulation of data or control transfer behavior of the fetched instruction in a manner incompatible with the architectural definition of the fetched instruction in the fetched instruction's native architecture.

75. (new 7/8/2002) The apparatus of claim 70, further comprising the steps of:

triggering a synchronous interrupt on fetch or execution of the fetched instruction; based at least in part on a memory state of the computer and the address of the fetched instruction, wherein the architectural definition of the fetched instruction in the fetched instruction's native architecture does not call for an interrupt.

76. (new 7/8/2002) The apparatus of claim 70, wherein:
the table entries are indexed by virtual addresses of instructions.

77. (new 7/8/2002) The apparatus of claim 70, wherein:
the table entries are indexed by physical addresses of instructions.

78. (new 7/8/2002) The method of claim 9, wherein a portion of the memory state relevant to the interrupt trigger includes a content of the table entry indexed by the address of the instruction, wherein the architectural definition of the instruction does not call for an interrupt.

79. (new 7/8/2002) The method of claim 2, wherein:
the controlling of instruction behavior includes altering a manipulation of data or control transfer [of control] behavior of the instruction in a manner incompatible with the architectural definition of the instruction in the instruction's native architecture.

80. (new 7/8/2002) The method of claim 2, wherein:
the step of consulting the table as part of executing an instruction is performed as part of fetching the instruction.

G

STRUCTURED COMPUTER ORGANIZATION

SECOND EDITION

ANDREW S. TANENBAUM

Vrije Universiteit
Amsterdam, The Netherlands

PRENTICE-HALL, INC.

ENGLEWOOD CLIFFS, NEW JERSEY 07632

G

Library of Congress Cataloging in Publication Data

Tanenbaum, Andrew, S. (date)
Structured Computer Organization

Includes index.
1. Electronic digital computers—Programming.
I. Title
QA76.6.T38 1984 001.64'2 83-2916
ISBN 0-13-854489-1

To Suzanne, Barbara, Marvin and the memory of Sweetie π

Production/editorial supervision: Nancy Milnamow
Manufacturing buyer: Gordon Osbourne

10 9 8 7 6 5

Printed in the United States of America

ISBN 0-13-854489-1

PRENTICE-HALL INTERNATIONAL, INC., *London*
PRENTICE-HALL OF AUSTRALIA PTY. LTD., *Sydney*
EDITORA PRENTICE-HALL DO BRASIL, LTDA., *Rio de Janeiro*
PRENTICE-HALL OF CANADA, LTD., *Toronto*
PRENTICE-HALL OF INDIA PRIVATE LTD., *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*
PRENTICE-HALL OF SOUTHEAST ASIA PTE. LTD., *Singapore*
WHITEHALL BOOKS LTD., *Wellington, New Zealand*

G

5

THE CONVENTIONAL MACHINE LEVEL

This chapter introduces the conventional machine level (level 2) and discusses many aspects of its architecture. Historically, level 2 was developed before any of the other levels, and it is still widely (and incorrectly) regarded as "the" machine language. This situation has come about because on many machines the microprogram is in a read-only memory, which means that users (as opposed to the machine's manufacturer) cannot write programs for level 1. Furthermore, even on machines that are user microprogrammable, the enormous complexity of the level 1 architecture is enough to scare off all but the most stouthearted programmers. In addition, because no machines have protection hardware at level 1, it is not possible to allow one person to debug new microprograms while anyone else is using the machine. This characteristic further inhibits user microprogramming.

5.1. EXAMPLES OF THE CONVENTIONAL MACHINE LEVEL

Rather than attempt to define rigorously what the conventional machine level is (which is probably impossible anyway), we will introduce this level by means of four examples. The next four sections are devoted to examining the conventional machine level of four families of well-known, commercially available computers: the IBM 370, the DEC PDP-11, the Motorola MC68000, and the Zilog Z80. The purpose of choosing four existing computers to study is to show how the ideas discussed here can be applied to the "real world." These machines will be compared and contrasted in many ways and they will continue to serve as running examples in succeeding chapters, as they have in past ones.

You should not draw the conclusion that the remainder of the book is about programming the 370, PDP-11, 68000 or Z80. These machines will be used to illustrate the idea of designing a computer as a series of levels. Various features of their respective organizations will be examined and some information about programming them will be introduced where necessary. Early in the chapter, the complete instruction sets for all four machines will be presented in tables. You are not expected to fully understand them initially, although going through each list and making educated guesses about what the instructions probably do is certainly instructive. Many of the instructions will be discussed in more detail later in the chapter.

Nevertheless, you should keep in mind the central, unifying idea that computers can be designed in a structured way. The technique of building a computer as a series of levels is a powerful structuring technique. An understanding of some of the details and idiosyncrasies of the four machines is necessary to understand the various levels but try to relate the details to the overall structure and do not wallow in them.

5.1.1. IBM System/370

In 1964 IBM introduced the System/360, a family of computers with identical level 2 architectures and instruction sets but spanning a wide range of performance and price. The idea behind the 360 was to allow customers to buy whichever model was appropriate at the time of purchase and later be able to upgrade to a larger model as the work increased, without having to rewrite any programs. In the early 1970s, IBM brought out various models of the System/370 series as successors to the 360 series using more modern technology. The level 2 architecture of the 370 series is a minor extension of the 360 series. In subsequent years IBM marketed the 43xx series (4331, 4341, etc.), 30xx series (3031, 3032, 3033, etc.) and other computers whose level 2 architectures are practically identical to that of the 370. For the sake of simplicity, we have chosen to refer to these machines collectively as the 370, but you should be aware that minor architectural differences exist among the various series, and even between models of one series. Thus at level 2, all the machines are nearly identical but at level 0 and level 1 they are all completely different.

An IBM 370 consists of one or more CPUs, one or more I/O processors, a main memory, and various I/O devices, as shown in Fig. 5-1. Three kinds of I/O processors exist, called **multiplexer channels**, **block multiplexer channels**, and **selector channels**, the first type being used with low-speed I/O devices, such as card readers, printers, and card punches, and the other two being used with high-speed I/O devices, such as disks, drums, and tapes. All the processors in the computer have access to main memory for reading and writing. The channels have a few internal registers but no main memory of their own.

The smallest addressable unit in the main memory is the byte, consisting of 8 bits. Each byte has a unique address, numbered 0, 1, 2, 3, 4, ..., $n - 1$, where n is the number of bytes of memory, up to a maximum of $n = 2^{24}$. Two consecutive

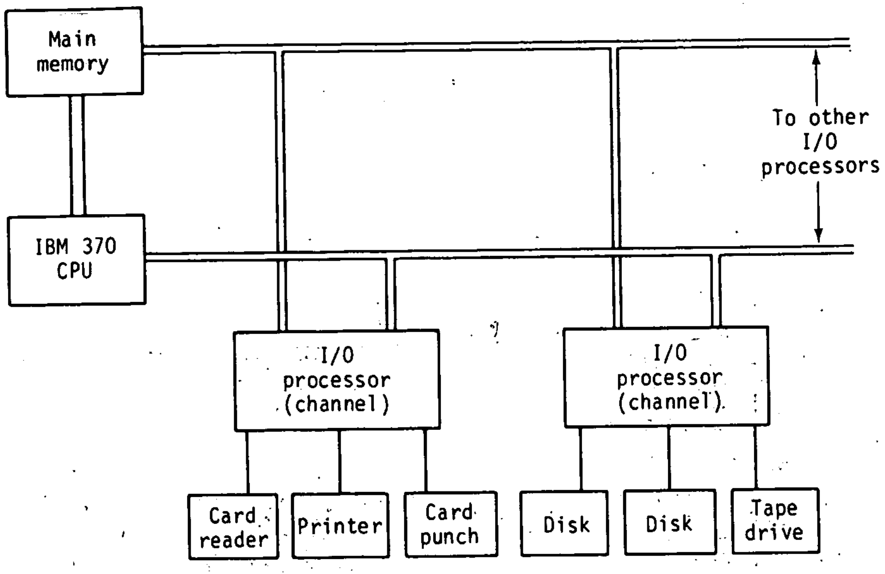


Fig. 5-1. Organization of an IBM 370 computer with one CPU and two I/O processors (channels).

bytes form a half word, 4 consecutive bytes form a word, and 8 consecutive bytes form a double word. Words are more important than half words or double words, so the 370 is often regarded as having a 32-bit word. The address of a half word, word, or double word is the address of its lowest-numbered byte, which, on some older models, must be an integral multiple of 2, 4, or 8, respectively. Figure 5-2 illustrates the addressing structure of the 370 memory. Each byte is part of a half word, a word, and a double word. For example, byte 7 is the low-order (rightmost) byte of the half word at 6, the word at 4, and the double word at 0. The CPU has instructions for fetching bytes, half words, words, and double words.

The 370 has a special format for storing packed decimal numbers. Four bits are needed to represent a digit in the range 0 to 9, so two decimal digits can be packed into one 8-bit byte. In this format, a byte may contain any number from 0 to 99. If used to store numbers in pure binary form, a byte can hold any number between 0 and 255, and 7 bits are sufficient to hold all the numbers from 0 to 99. The packed decimal format does not use memory optimally.

Packed decimal numbers do have certain advantages over binary numbers, however. Data input to the computer or output from the computer are in decimal notation, because people use decimal numbers. When binary numbers are used internally, the input must be converted from decimal to binary, processed in binary, and then reconverted from binary to decimal. If the amount of computation is small, the CPU may spend most of its time performing conversions. If the amount of computation is

6

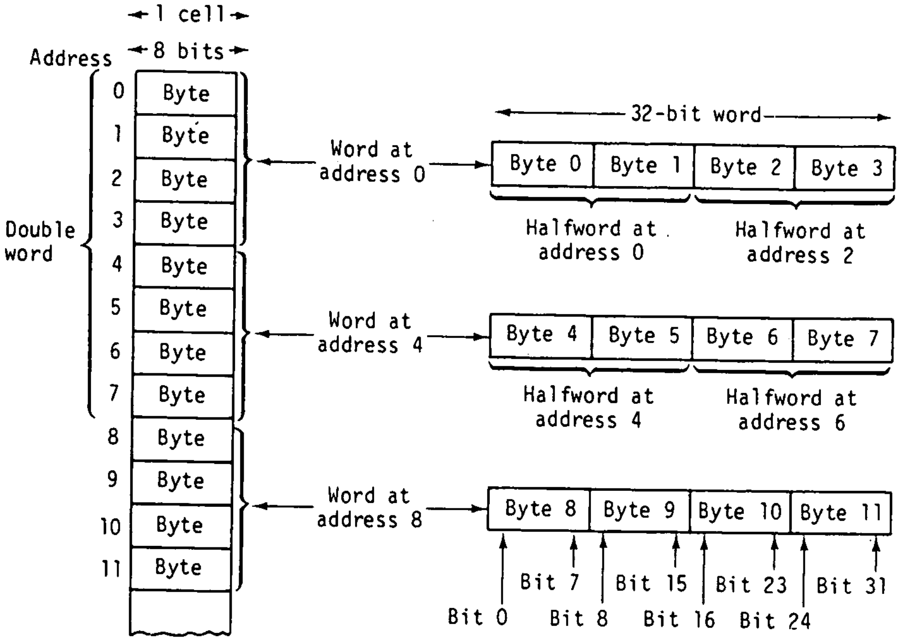


Fig. 5-2. Addressing structure of the IBM 370's main memory.

large, the faster speed of the binary arithmetic instructions makes the conversions worthwhile. Packed decimal numbers are widely used in business applications.

A program at the conventional machine level on a 370 has access to 20 high-speed CPU registers used for performing arithmetic and logical operations, as well as for storing intermediate results. Sixteen of these are general-purpose registers of length 32 bits, numbered from 0 to 15. The other four registers, numbered 0, 2, 4, and 6, are 64 bits long and are used for floating-point arithmetic. The 370 hardware also contains a number of other registers, such as an instruction register, program status word, MAR, and MBR but they are only accessible at the microprogramming level. Figure 5-3 illustrates the registers used by conventional-machine-level programs. The 370 also has 16 control registers, but these are used only by the operating system and will not concern us further.

The 370 level 2 instructions are either 16, 32, or 48 bits in length and may be located at any even address. Nearly all instructions contain an 8-bit operation code, specifying which operation is to be carried out. The remaining bits are used to specify where the data for the instruction is located—in registers, memory, or both. The 370 has some general-purpose instructions used in almost all programs, some instructions intended primarily for scientific calculations, and some instructions primarily useful for commercial applications. The level 2 instruction set contains about 200 instructions. A list of most of these instructions is given in Fig. 5-4.

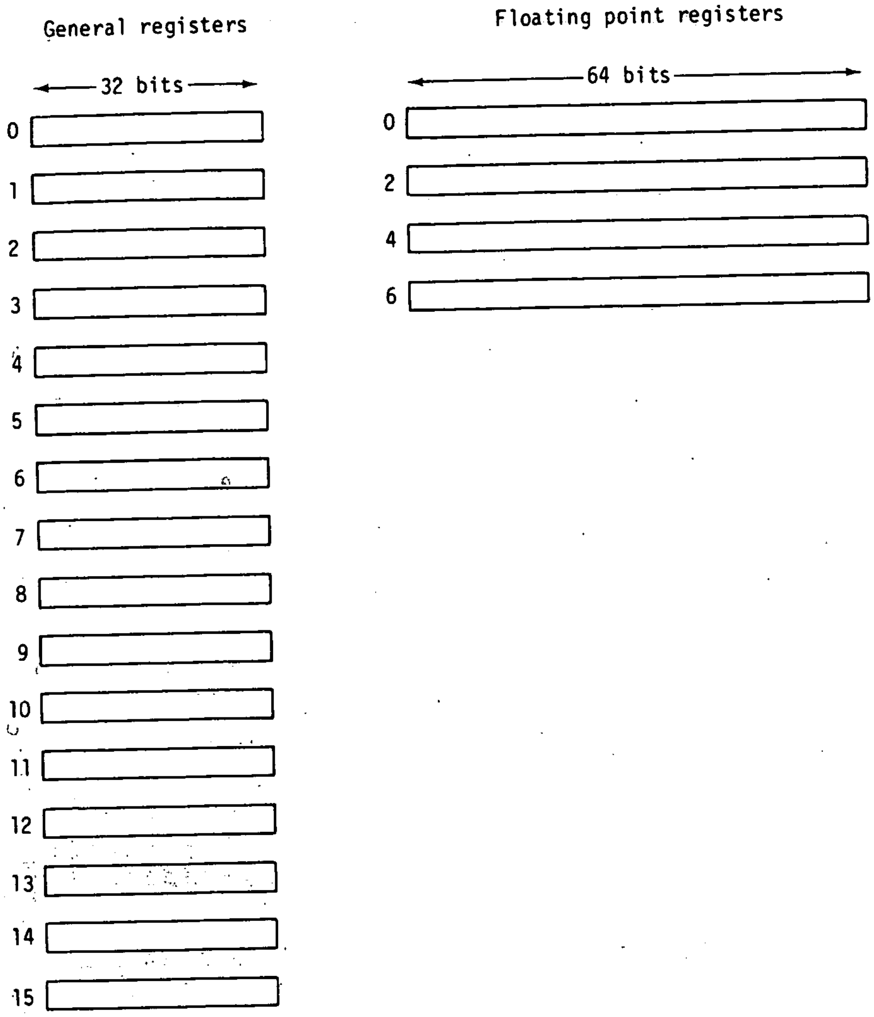


Fig. 5-3. General registers and floating-point registers on the 370.

5.1.2. DEC PDP-11

The DEC PDP-11 series consists of a number of small to medium-sized computers. Due to their short (16-bit) word length, they are often called **minicomputers**, although, as we mentioned earlier, the boundaries between mainframes, minicomputers, and microcomputers are highly elusive. The PDP-11s are widely used in such applications as data communication, industrial process control, scientific experiment monitoring and data collection, interactive computer graphics, and education.

A PDP-11 consists of a CPU, a main memory, and various I/O devices, as

RR format

xxxx	Branching and status switching		Fixed-point fullword and logical		Floating-point long		Floating-point short	
	0000xxxx		0001xxxx		0010xxxx		0011xxxx	
0000			LPR	Load positive	LPDR	Load positive	LPER	Load positive
0001			LNR	Load negative	LNDR	Load negative	LNER	Load negative
0010			LTR	Load and test	LTDR	Load and test	LTER	Load and test
0011			LCR	Load complement	LCDR	Load complement	LCER	Load complement
0100	SPM	Set program mask	NR	And	HDR	Halve	HER	Halve
0101	BALR	Branch and link	CLR	Compare logical	LRDR	Load rounded (E to L)	LRER	Load rounded (L to SH)
0110	BCTR	Branch on count	OR	Or	MXR	Multiply (E)	AXR	Add normalized (E)
0111	BCR	Branch/condition	XR	Exclusive or	MXDR	Multiply (L to E)	SXR	Subtract normalized (E)
1000	SSK	Set storage key	LR	Load	LDR	Load	LER	Load
1001	ISK	Insert key	CR	Compare	CDR	Compare	CER	Compare
1010	SVC	Supervisor call	AR	Add	ADR	Add N	AER	Add N
1011			SR	Subtract	SDR	Subtract N	SER	Subtract N
1100			MR	Multiply	MDR	Multiply	MER	Multiply
1101			DR	Divide	DDR	Divide	DER	Divide
1110	MVCL	Move long	ALR	Add logical	AWR	Add U	AUR	Add U
1111	CLCL	Compare logical long	SLR	Subtract logical	SWR	Subtract U	SUR	Subtract U

RX format

xxxx	Fixed-point halfword and branching		Fixed-point fullword and logical		Floating-point long		Floating-point short	
	0100xxxx		0101xxxx		0110xxxx		0111xxxx	
0000	STH	Store	ST	Store	STD	Store	STE	Store
0001	LA	Load address						
0010	STC	Store character						
0011	IC	Insert character						
0100	EX	Execute	N	And				
0101	BAL	Branch and link	CL	Compare logical				
0110	BCT	Branch on count	O	Or				
0111	BC	Branch/condition	X	Exclusive or	MXD	Multiply (L to E)		
1000	LH	Load	L	Load	LD	Load	LE	Load
1001	CH	Compare	C	Compare	CD	Compare	CE	Compare
1010	AH	Add	A	Add	AD	Add N	AE	Add N
1011	SH	Subtract	S	Subtract	SD	Subtract N	SE	Subtract N
1100	MH	Multiply	M	Multiply	MD	Multiply	ME	Multiply
1101			D	Divide	DD	Divide	DE	Divide
1110	CVD	Convert-decimal	AL	Add logical	AW	Add U	AU	Add U
1111	CVB	Convert-binary	SL	Subtract logical	SW	Subtract U	SU	Subtract U

Fig. 5-4. The IBM 370 Instruction set. The instructions marked with an asterisk are actually groups of instructions with a 16-bit opcode. The abbreviation for each instruction is given before the name.

RS, SI, S format

	Branching status switching and shifting		Fixed-point logical and input/output					
xxxx	1000xxxx		1001xxxx		1010xxxx		1011xxxx	
0000	SSM	Set system mask	STM	Store multiple				
0001			TM	Test under mask			LRA	Load real address
0010	LPSW	Load PSW	MVI	Move				(15 instructions)*
0011		diagnose	TS	Test and set				
0100	WRD	Write direct	NI	And				
0101	RDD	Read direct	CLI	Compare logical				
0110	BXH	Branch/high	OI	Or			STCTL	Store control
0111	BXLE	Branch/low-eoual	XI	Exclusive or			LCTL	Load control
1000	SRL	Shift right SL	LM	Load multiple				
1001	SLL	Shift left SL						
1010	SRA	Shift right single					CS	Compare and swap
1011	SLA	Shift left single					CDS	Compare D and swap
1100	SRDL	Shift right DL	SIO	Start I/O *	STNSM	Store then and mask		
1101	SLDL	Shift left DL	TIO	Test I/O *	STOSM	Store then or mask	CLM	Comp chars masked
1110	SRDA	Shift right double	HIO	Halt I/O *	SIGP	Signal processor	STCM	Store chars masked
1111	SLDA	Shift left double	TCH	Test channel	MC	Monitor call	ICM	Insert chars masked

SS format

			Logical				Decimal	
xxxx	1100xxxx		1101xxxx		1110xxxx		1111xxxx	
0000							SRP	Shift and round
0001			MVN	Move numerics			MVO	Move with offset
0010			MVC	Move			PACK	Pack
0011			MVZ	Move zones			UNPK	Unpack
0100			NC	And				
0101			CLC	Compare logical				
0110			OC	Or				
0111			XC	Exclusive or				
1000							ZAP	Zero and add
1001							CP	Compare
1010							AP	Add
1011							SP	Subtract
1100			TR	Translate			MP	Multiply
1101			TRT	Translate and test			DP	Divide
1110			ED	Edit				
1111			EDMK	Edit and mark				

Note: N = Normalized
SL = Single logical

DL = Double logical
U = Unnormalized

E = Extended
L = Long
SH = Short

More complicated I/O devices require more status registers. The RL01/RL02 disk, for example, uses a total of four registers containing a variety of fields, as shown in Fig. 5-39(b). Each register has a unique memory address by which the program can read it or store into it. More sophisticated disks can have more than a dozen device registers with more than 100 fields. The RL01/RL02 control register provides status about the controller and drives, including error reporting. It also has a field which the program loads with a function code to indicate the operation desired. The meaning of the other three registers depends on the function code loaded. For READ and WRITE, the other three contain the memory address to read into or write from, the disk address, and the word count. For SEEK, WRITE CHECK, and other operations, they have somewhat different meanings.

The Z80 has explicit I/O instructions, although system designers are free to do I/O with memory mapping if they choose, as we did in Chap. 3. When memory mapping is not used, the Z80 still has (8-bit-wide) device registers, only now they are not part of the memory address space. Instead, each device register has a number from 0 to 255, called an **I/O port**.

The simplest I/O instructions are IN A,(N), which copies one byte from port N to the A register, and OUT (N),A, which copies one byte from the A register to port N. The byte transferred may be either data, control information, or status information, depending on the port selected. Slightly more complex are IN R,(C) and OUT (C),R, which take the port number from the C register. The next step up are INI, IND, OUTI, and OUTD. All these instructions expect a memory address in HL, a count in B, and a port number in C. First, a normal IN or OUT is done, copying a byte to or from the memory location pointed to by HL. Then HL is incremented or decremented by 1 and B is decremented by 1. The final I/O instructions are INIR, INDR, OTIR, and OTDR, which are the same as the previous four, except that they keep going until B = 0. In effect, each one does a block transfer to or from memory. This transfer is not DMA, however, because the CPU is occupied the entire time. Nevertheless, it is faster and takes up less space in memory than an explicitly programmed loop to do the same job.

5.5. FLOW OF CONTROL

Flow of control refers to the sequence in which instructions are executed. In general, successively executed instructions are fetched from consecutive memory locations. Procedure calls cause the flow of control to be altered, stopping the procedure currently executing and starting the called procedure. Coroutines are related to procedures and cause similar alterations in the flow of control. Traps and interrupts also cause the flow of control to be altered when special conditions occur. All these topics will be discussed in the following sections.

5.5.1. Sequential Flow of Control and Jumps

Most instructions do not alter the flow of control. After an instruction is executed, the one following it in memory is fetched and executed. After each instruction, the program counter is increased by the number of memory locations in that instruction. If observed over an interval of time that is long compared to the average instruction time, the program counter is approximately a linear function of time, increasing by the average instruction length per average instruction time. Stated another way, the dynamic order in which the processor actually executes the instructions is the same as the order in which they appear on the program listing.

If a program contains jumps, this simple relation between the order in which instructions appear in memory and the order in which they are executed is no longer true. When jumps are present, the program counter is no longer a monotonically increasing function of time, as shown in Fig. 5-40(b). As a result, it becomes difficult to visualize the instruction execution sequence from the program listing. When programmers have trouble keeping track of the sequence in which the processor will execute the instructions, they are prone to make errors. This observation led Dijkstra (1968a) to write a then controversial letter entitled "GO TO Statement Considered Harmful," in which he suggested avoiding GO TO statements. Since that time languages without GO TO statements have become popular. Of course, these programs compile down to level 2 programs that may contain many jumps, because the implementation of IF, WHILE, and other high-level control structures require jumping around.

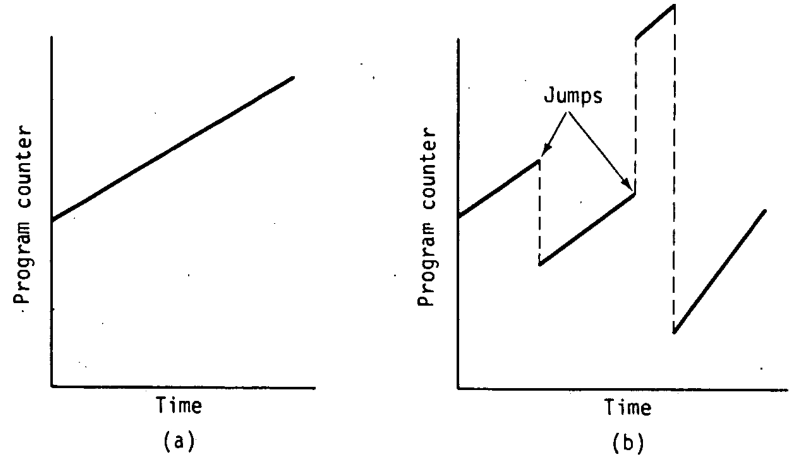


Fig. 5-40. Program counter as a function of time (smoothed). (a) Without jumps. (b) With jumps.

Jumps are frequently difficult to avoid when programming in a language lacking structuring statements such as IF ... THEN ... ELSE and WHILE ... DO .... Relying on jumps as the primary method of controlling the flow of execution makes it difficult, if not impossible, to write error-free, well-structured programs. For this and other reasons, programming at the conventional machine level is becoming obsolete.

5.5.2. Procedures

The most important technique for structuring programs is the procedure. From one point of view, a procedure call alters the flow of control just as a jump does, but unlike the jump, when finished performing its task, it returns control to the statement or instruction following the call.

However, from another point of view, a procedure body can be regarded as defining a new instruction on a higher level. From this standpoint, a procedure call can be thought of as a single instruction, even though the procedure may be quite complicated. Similarly, a person programming at level 2 can certainly regard the multiplication instruction as a single instruction, even though it is carried out by an interpreter running at level 1 as a large number of successive steps.

By writing a collection of procedures, a programmer can define a new level with a new, larger, and more convenient instruction set. Programs for this new level consist of sequences of instructions, some of which are procedure calls and some of which are the original level 2 instructions. Associated with the execution of a program at this new level is a "virtual program counter," which points to the current instruction (counting a procedure execution as a single instruction) and increases monotonically in time. The direct correspondence between the execution sequence and the listing sequence makes it easy to understand what the program does.

In Sec. 5.4.5, we mentioned recursive procedures—that is, procedures that call themselves. Now we will give an example of one. The "Towers of Hanoi" is an ancient problem that has a simple solution involving recursion. The problem requires three pegs, on the first of which sit a series of n concentric disks, each of which is smaller in diameter than the disk directly below it. The second and third pegs are initially empty. The object is to transfer all the disks to peg 3, one disk at a time, but at no time may a larger disk rest on a smaller one. Figure 5-41 shows the initial configuration for $n = 5$ disks.

The solution of moving n disks from peg 1 to peg 3 consists first of moving $n - 1$ disks from peg 1 to peg 2, then moving 1 disk from peg 1 to peg 3, then moving $n - 1$ disks from peg 2 to peg 3 (see Fig. 5-42). To solve the problem we need a procedure to move k disks from peg i to peg j. Whenever this procedure is called, by

$$towers(n,\ i,\ j)$$

the solution is printed out. The procedure first tests to see if $n = 1$. If so, the solution is trivial, just move the one disk from i to j. If $n \neq 1$, the solution consists of three parts as discussed above, each being a recursive procedure call.

5.5.4. Traps

A **trap** is a kind of automatic procedure call initiated by some condition caused by the program, usually an important but rarely occurring condition. A good example is overflow. On many computers, if the result of an arithmetic operation exceeds the largest number that can be represented, a trap occurs, meaning that the flow of control is switched to some fixed memory location instead of continuing in sequence. At that fixed location is a jump to a procedure called the overflow trap handler, which performs some appropriate action, such as printing an error message. If the result of an operation is within range, no trap occurs.

The essential point about a trap is that it is initiated by some exceptional condition caused by the program itself and detected by the hardware or microprogram. An alternative method of handling overflow is to have a 1-bit register that is set to 1 whenever an overflow occurs. A programmer who wants to check for overflow must include an explicit "jump if overflow bit is set" instruction after every arithmetic instruction. Doing so would be both slow and wasteful of space. Traps save both time and memory compared with explicit programmer controlled checking.

The trap may be implemented by an explicit test performed by the interpreter at level 1. If an overflow is detected, the trap address is loaded into the program counter. What is a trap at one level may be under program control at a lower level. Having the microprogram make the test still saves time compared to a programmer test, because it can be easily overlapped with something else. It also saves memory, because it need only occur in a few level 1 procedures, independent of how many arithmetic instructions occur in the main program.

A few common conditions that can cause traps are floating-point overflow, floating-point underflow, integer overflow, protection violation, undefined opcode, stack overflow, attempt to start nonexistent I/O device, attempt to fetch a word from an odd-numbered address and division by zero.

5.5.5. Interrupts

Interrupts are changes in the flow of control caused not by the running program but by something else, usually related to I/O. For example, a program may instruct the disk to start transferring information, and set the disk up to provide an interrupt as soon as the transfer is finished. Like the trap, the interrupt stops the running program and transfers control to an interrupt handler, which performs some appropriate action. When finished, the interrupt handler returns control to the interrupted program. It must restart the interrupted process in exactly the same state it was in when the interrupt occurred, which means restoring all the internal registers to their preinterrupt state.

The essential difference between traps and interrupts is this: *traps* are synchronous with the program and *interrupts* are asynchronous. If the program is rerun a million times with the same input, the traps will reoccur in the same place each time but the interrupts may vary, depending, for example, on precisely when a person at a

G

terminal pushes the carriage return key. The reason for the reproducibility of traps and irreproducibility of interrupts is that traps are caused directly by the program and interrupts are, at best, indirectly caused by the program.

The need for interrupts arises when input or output can proceed in parallel with CPU execution. On computers where the CPU issues an I/O instruction and then stops to wait for the I/O to be completed, there is no need for an interrupt. When the I/O is finished, the CPU is automatically restarted at the instruction following the I/O instruction. Because the CPU can generally execute many thousands of instructions during the time required to complete a single I/O instruction, it is wasteful to force the CPU to be idle during this time. Interrupt schemes allow the CPU to compute concurrently with the I/O and be signaled as soon as the I/O is completed.

A large computer may have many I/O devices running at the same time. For example, it might be reading data from cards, printing results on the line printer, writing output on a disk for future use, and plotting a graph of results on the plotter. All this activity can lead to complicated situations. When the card reader has finished reading a card, the CPU is interrupted and the card reader service procedure is begun. The card reader service procedure must move the card just read to the main memory location where the CPU expects it (if it is not already there), check to see if any reading errors occurred, possibly check to see if each card column contains a valid character, and issue an instruction to start reading the next card.

A nonzero probability exists that another I/O device—for example, the disk—will complete its I/O instruction before the reader service procedure has completed its task. This situation can be handled in one of two ways. First, the disk can cause a CPU interrupt, halting execution of the card reader service procedure and starting execution of the disk service procedure. Second, the disk can be forced to wait until the reader service procedure is finished and can then cause an interrupt. We will now examine these possibilities in detail.

If we allow the disk to interrupt the card reader service procedure, we must also be prepared for the printer to interrupt the disk service procedure and for the plotter to interrupt the printer service procedure. If this interrupt sequence actually occurs, it is necessary to decide what to do when the plotter service procedure finishes. Possibilities are to continue the printer, disk, or card reader service procedures or to continue the CPU program that was running when the card reader interrupt occurred. It is clear that the administration involved in keeping track of which procedure to run when can get complicated.

One method for simplifying this administration is to require that all interrupts be **transparent**, which means that whenever an interrupt occurs, the state of the interrupted process is saved, including the program counter, registers, and condition codes. The interrupt service procedure is then run. Finally, the state of the interrupted process is restored to exactly the same condition it was in when the interrupt occurred and the process restarted.

The interrupted process neither requires any special precautions nor needs to be concerned with the interrupt handling. It is not even aware of its existence (unless it is timing something). Because the program running at the time of the interrupt is not aware of the fact that it has been interrupted, stopped, and later restarted, the interrupt is said to be transparent (or invisible). If all interrupts are transparent, an interrupt procedure will not even notice if it, itself, is interrupted.

Turning again to our earlier example, it is clear that the card reader interrupt must be transparent to the main program, the disk interrupt must be transparent to the card reader service procedure, and so on. When the plotter service procedure finally finishes its task, the printer service procedure must be restarted, not one of the other service procedures.

Similarly, for the printer service procedure to be transparent to the disk service procedure, the latter must be restarted (when the printer service procedure is through) from the point where it was interrupted. In other words, the interrupt service procedures must be restarted in the reverse order in which they occurred, as shown in Fig. 5-49. Whenever an interrupt service procedure completes its task, the most recently interrupted procedure must be restarted.

From Fig. 5-49 we see that interrupts are nested in time, meaning that a program will not be restarted until all the interrupts subsequent to it have been completely processed. Situations involving nesting are common in computer science. All nesting situations have one property in common: an inner nest is always completed before the surrounding nest is completed. A stack can often be used to implement a nesting situation. Whenever a new nest is entered, the state of the computation just before the entry is saved on the stack. Whenever a nest is exited, the state of the computation just previous to entering that nest is popped off the stack and restored.

Figure 5-50 illustrates the use of a stack for processing interrupts. The numbers 1 to 9 represent the time intervals shown in Fig. 5-49. During interval 1, no preceding state need be remembered, and the stack is empty. After the card reader interrupt has occurred, the state of the main program at the time of the interrupt must be remembered so that it can be restarted in the correct place later. This situation is shown as 2. When the disk interrupts the card reader service procedure, the state of the card reader service procedure must also be stacked, shown as 3.

Each of the nine stack configurations refers to some sequence of as yet uncompleted interrupt procedures. If another interrupt occurs at 7, while the disk service procedure is running, the state of that procedure will be saved again. If the computer possesses many I/O devices, a given interrupt service procedure may be stopped and resumed several times before it completes.

The model we have just given for interrupt processing is, however, not complete because we have ignored the critical timing aspects of I/O devices. Some I/O devices must be serviced within a specific time interval or information will be lost. For example, if data are being transmitted over a communication line at 960 characters/sec, a character arrives in the receiver buffer every 1042 μsec. If the interrupt service procedure fails to fetch the character within 1042 μsec, it may be overwritten by the next one and lost. An interrupt system needs a provision for handling this kind of problem. In other words, once the service procedure for a highly critical interrupt has begun, it must not be interrupted by a less critical interrupt.

When an I/O processor on the 370 finishes executing its program, it can interrupt

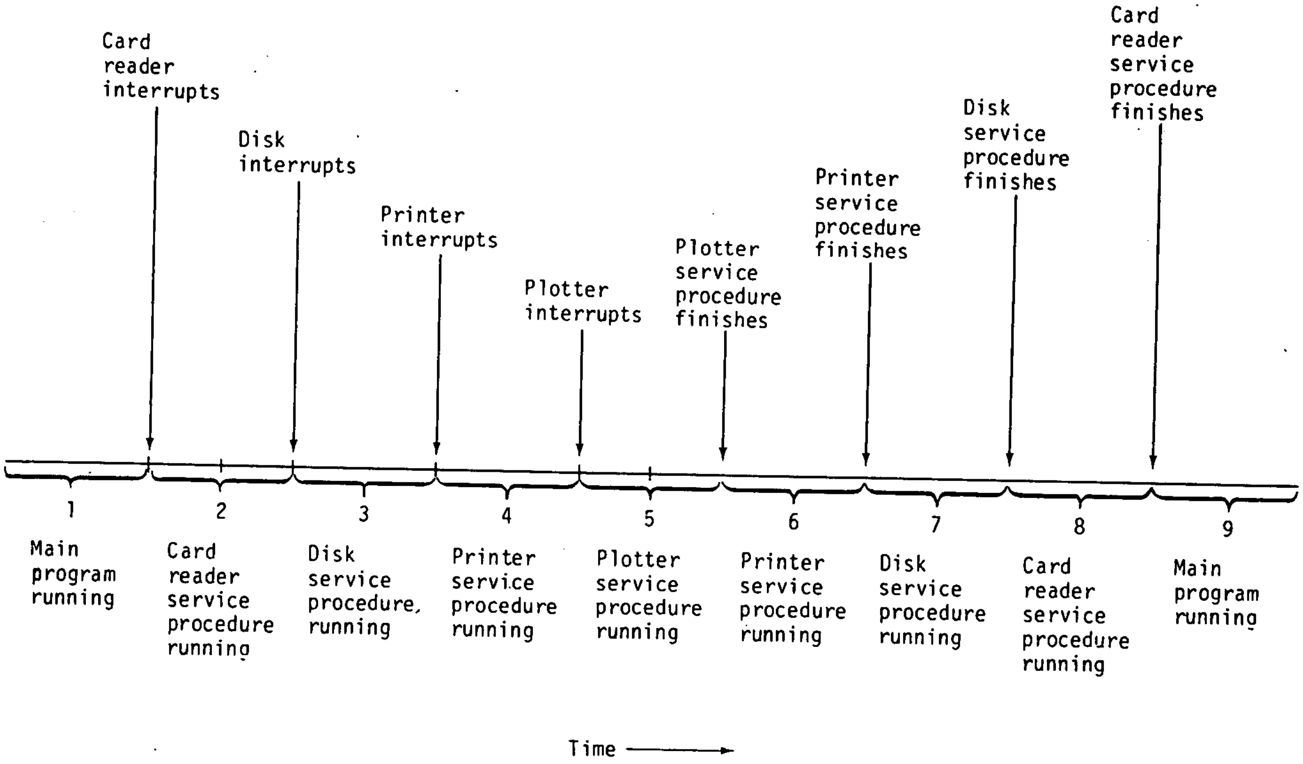

Card reader interrupts
Disk interrupts
Printer interrupts
Plotter interrupts
Plotter service procedure finishes
Printer service procedure finishes
Disk service procedure finishes
Card reader service procedure finishes
1
2
3
4
5
6
7
8
9
Main program running
Card reader service procedure running
Disk service procedure, running
Printer service procedure running
Plotter service procedure running
Printer service procedure running
Disk service procedure running
Card reader service procedure running
Main program running
Time

Fig. 5-49. Time sequence of interrupts from the card reader, disk, printer, and plotter.

9

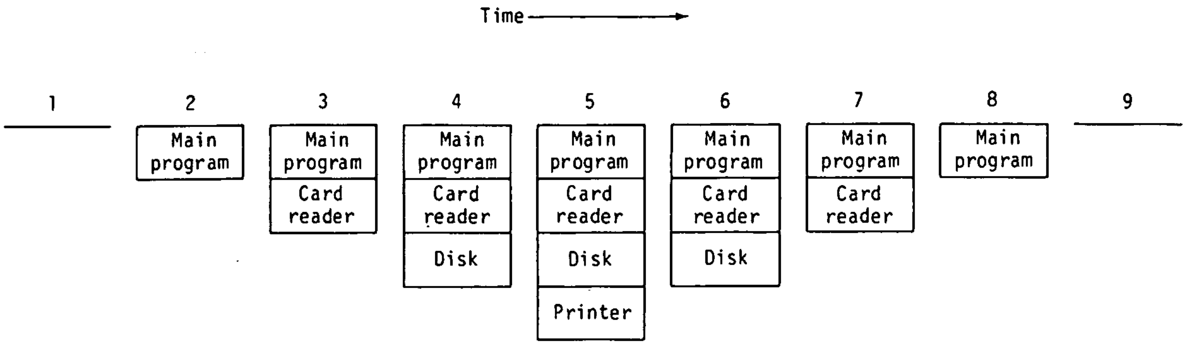


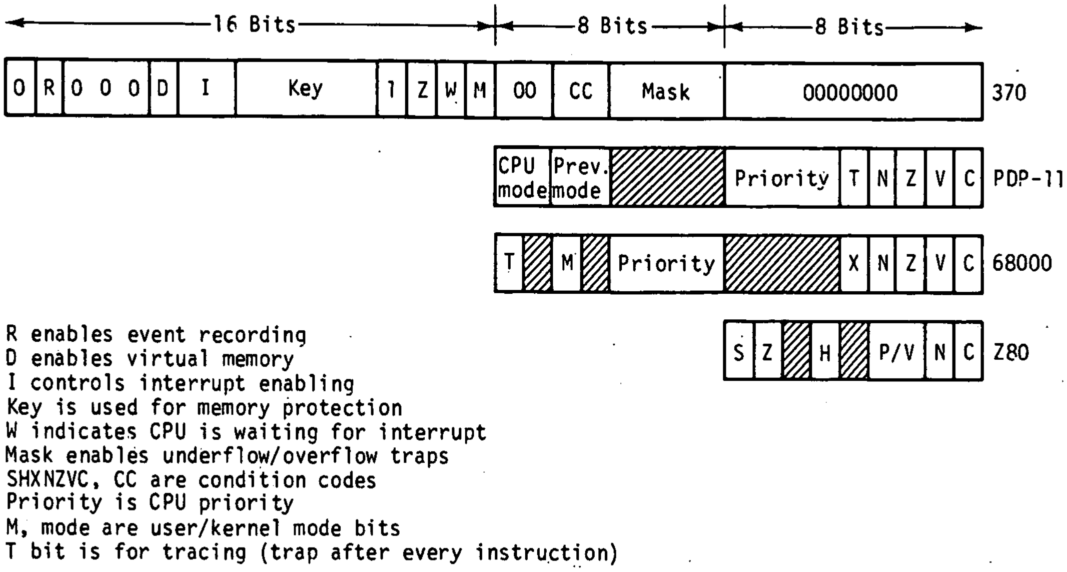
Fig. 5-50. Use of a stack for interrupt handling. Each box is the saved state of the indicated process. The numbers correspond to Fig. 5-49. The stack grows downward.

the CPU. It does so in the following steps. The CPU has a 64-bit register called a **program status word** (PSW) that contains the program counter, condition code, interrupt code, and other pieces of status information, as illustrated in Fig. 5-51. When an interrupt occurs, the PSW's interrupt code is set to the number of the interrupting device. Then the PSW is stored in location 56 and a new PSW is loaded from location 120. As soon as the new PSW has been loaded, the CPU begins execution at the start of a general interrupt service procedure. The interrupt service procedure must first determine which I/O device finished by examining the interrupt code in the old PSW at location 56. Then it can call the appropriate service procedure.

If a second interrupt occurred while the first one was being processed, the current PSW would be stored at location 56, thereby erasing the one already there. The situation is similar to a procedure call instruction that always puts the return address in a fixed place in memory. To prevent having a PSW overwritten before the interrupt procedure has had a chance to save it, the 370 has a bit associated with each I/O processor called a **mask bit**. When this bit is a 0, interrupts from that I/O processor are forced to wait until it becomes a 1. Setting mask bits to zero is called **disabling** interrupts. The mask bits are located in an internal processor register. If the *I* bit in Fig. 5-51 is 0, all interrupts are disabled, no matter what values the mask bits have.

When the PDP-11 is interrupted by an I/O device, the PSW (see Fig. 5-51) and program counter are pushed onto the stack, and a new PSW and program counter are loaded from the memory address associated with the I/O device. These memory addresses are called **interrupt vectors** and each device has a unique one. During the hardware interrupt sequence, the device specifies an interrupt vector by putting the vector's address on the UNIBUS. Each interrupt vector contains the starting address of the service procedure for the corresponding device, thus eliminating the need to test which device caused the interrupt. The 370 has only one interrupt vector for all I/O devices, so the interrupt handler must first determine which device wants attention. Only then can it call the proper service procedure.

G7



R enables event recording
D enables virtual memory
I controls interrupt enabling
Key is used for memory protection
W indicates CPU is waiting for interrupt
Mask enables underflow/overflow traps
SHXNZVC, CC are condition codes
Priority is CPU priority
M, mode are user/kernel mode bits
T bit is for tracing (trap after every instruction)

Fig. 5-51. The PSW on four computers. The 370 PSW is the EC format and only the upper 32 bits are shown. The lower 32 bits contain eight zeros and the program counter.

The PDP-11 has a system of priority interrupts. Each I/O device has a priority number associated with it. The CPU also has a priority (from 0 to 7), which can be set by the program, and which is part of the PSW. If the priority of the I/O device is higher than the current CPU priority, the interrupt takes place; otherwise, it is forced to wait until the CPU priority is reduced. This feature can be used to ensure that time-critical interrupt service procedures can be interrupted only by still more critical ones, not by less critical ones. For example, if the disk interrupt service procedure runs at priority 5, an interrupt from magnetic tape at priority 6 can interrupt it but an interrupt from the paper tape reader at priority 4 will be forced to wait until the CPU priority is set to 3 or less, which will not occur until the critical disk service procedure is finished. The priority of an I/O device is determined by a switch on the device itself. The devices that need the fastest service are naturally given the highest priorities. Traps use interrupt vectors, the same as true interrupts.

The 68000's interrupt system is similar to the PDP-11's, including the eight priority levels. When an external device whose priority is higher than the CPU's signals an interrupt, the PSW and program counter are stacked and a new program counter is fetched from the interrupt vector. A new PSW is not fetched from memory as on the PDP-11 but the CPU priority is set to that of the interrupting device. This approach saves space in the interrupt vectors because no PSWs need be stored but means that a priority n device service routine can be interrupted by a priority $n + 1$ device before the routine has been able to execute even one instruction. On the PDP-11, the new PSW may contain priority 7 to prevent all other interrupts for a few instructions to allow the routine to do some initial work without being disturbed. Afterward the routine can lower the priority if it wants to.

On the 370, PDP-11, and 68000 but not the Z80, the CPU is always in one of two (on some PDP-11 models, three) modes or states. The more powerful of the two is called **kernel mode** or **supervisor state**. The less powerful is called **user mode** or something similar. In user mode, some instructions, principally those that do I/O or affect the current mode, are forbidden and cause traps to kernel mode. In kernel mode, everything is allowed. When these machines are used for multiprogramming (time sharing), the user programs are forced to run in user mode to prevent them from interfering with each other. The operating system, in contrast, runs in kernel mode so that it can control the whole machine. A bit or field in the PSW determines the current mode. When an interrupt occurs on the 370 or PDP-11, the new PSW fetched determines which mode the interrupt routine will run in. In practice it is always kernel mode. On the 68000, no new PSW is loaded on interrupt, so the hardware always switches directly into kernel mode.

The PDP-11 and 68000 have different hardware stack pointers for user mode and kernel mode. The kernel stack pointer points to the kernel stack, which is in an area of memory protected from user programs. When the interrupt hardware switches the CPU into kernel mode, it simultaneously switches stack pointers, so the program counter and PSW are saved on the (protected) kernel stack rather than on the user stack.

The Z80 interrupt system is more primitive than that of the PDP-11 and 68000. Two (rather than eight) interrupt levels are present: maskable and nonmaskable. The maskable interrupts can be disabled by the DI instructions; the nonmaskable interrupts cannot be disabled. The latter are frequently used for emergenciessuch as shutting down industrial process control equipment in the few milliseconds available after an impending power failure has been detected.

The interrupt sequence consists of storing the program counter on the stack and disabling maskable interrupts. The accumulator and flags must be saved in software with the PUSH AF instruction. Nonmaskable interrupts always force control to address 102. Three different (software selected) modes are available for maskable interrupts. In mode 0, the interrupting device provides the next instruction to be executed on the data bus. Normally, it is RST. In mode 1, control is forced to address 56. In mode 2, the high-order 8 bits of the interrupt service routine come from the I register; the device provides the low-order 8 bits on the data bus. This somewhat peculiar scheme is intimately related to the issue of 8080 compatibility.

5.6. SUMMARY

The conventional machine level is what most people think of as "machine language." At this level the machine has a byte- or word-oriented memory ranging

Computer Architecture A Quantitative Approach

David A. Patterson
UNIVERSITY OF CALIFORNIA AT BERKELEY

John L. Hennessy
STANFORD UNIVERSITY

With a Contribution by
David Goldberg
Xerox Palo Alto Research Center

MORGAN KAUFMANN PUBLISHERS, INC.
SAN MATEO, CALIFORNIA

9

Sponsoring Editor Bruce Spatz
Production Manager Shirley Jowell
Technical Writer Walker Cunningham
Text Design Gary Head
Cover Design David Lance Goines
Copy Editor Linda Medoff
Proofreader Paul Medoff
Computer Typesetting and Graphics Fifth Street Computer Services

Library of Congress Cataloging-in-Publication Data

Patterson, David A.
Computer architecture : a quantitative approach / David A.
Patterson, John L. Hennessy
p. cm.
Includes bibliographical references
ISBN 1-55880- 069-8
1. Computer architecture. 2. Electronic digital computers
–Design and construction. I. Hennessy, John L. II. Title.
QA76.9.A73P377 1990
004.2'2–dc20 89-85227
CIP

Morgan Kaufmann Publishers, Inc.
Editorial Office: 2929 Campus Drive. San Mateo, CA 94403
Order from: P.O. Box 50490, Palo Alto, CA 94303-9953

ADVICE, PRAISE, & ERRORS: Any correspondence related to this publication or intended for the authors should be addressed to the editorial offices of Morgan Kaufmann Publishers, Inc., Dept. P&H APE. Information regarding error sightings is encouraged. Any error sightings that are accepted for correction in subsequent printings will be rewarded by the authors with a payment of $1.00 (U.S.) per correction. Electronic mail can be sent to bugs@vsop.stanford.edu.

INSTRUCTOR SUPPORT: For information on classroom software and other instructor materials available to adopters, please contact the editorial offices of Morgan Kaufmann Publishers, Inc.

A n *Add the number in storage location n into the accumulator*

H n *Transfer the number in storage location n into the multiplier register.*

E n *If the number in the accumulator is greater than or equal to zero execute next the order which stands in storage location n; otherwise proceed serially.*

I n *Read the next row of holes on tape and place the resulting 5 digits in the least significant places of storage location n.*

Z *Stop the machine and ring the warning bell.*

Selection from the list of 18 machine instructions for the EDSAC from Wilkes and Renwick [1949]

G

3 Instruction Set Design: Alternatives and Principles

3.1 Introduction

In this chapter and the next we will concentrate on instruction set architecture—the portion of the machine visible to the programmer or compiler writer. This chapter introduces the wide variety of design alternatives with which the instruction set architect is presented. In particular, this chapter focuses on three topics. First, we present a taxonomy of instruction set alternatives and give some qualitative assessment of the advantages and disadvantages of various approaches. Second, we present and analyze some instruction set measurements that are largely independent of a specific instruction set. Finally, we address the issue of languages and compilers and their bearing on instruction set architecture. Before we discuss how to classify architectures, we need to say something about the instruction set measurement.

Throughout this chapter and the next, we will be examining a wide variety of architectural measurements. These measurements depend on the programs measured and on the compilers used in making the measurements. The results should not be interpreted as absolute, and you might see different data if you did the measurement with a different compiler or a different set of programs. The authors believe that the measurements shown in these chapters are reasonably indicative of a class of typical applications. The measurements are presented using a small set of benchmarks so that the data can be reasonably displayed, and so that the differences among programs can be seen. An architect for a new machine would want to analyze a **much larger** collection of programs to make his architectural decisions. All the measurements shown are *dynamic*—that is, the frequency of a measured event is weighed by the number of times that event occurs during execution of the measured program.

Now, we will begin exploring how instruction set architectures can be classified and analyzed.

3.2 Classifying Instruction Set Architectures

Instruction sets can be broadly classified along the five dimensions described in Figure 3.1, which are roughly ordered by the role they play in distinguishing instruction sets.

The type of internal storage in the CPU is the most basic differentiation, so we will focus on the alternatives for this portion of the architecture in this section. As shown in Figure 3.2, the major choices are a stack, an accumulator, or a set of registers. Operands may be named explicitly or implicitly: The operands in a *stack architecture* are implicitly on the top of the stack; in an *accumulator architecture* one operand is implicitly the accumulator. *General-purpose register architectures* have only explicit operands—either registers or memory locations. Depending on the architecture, the explicit operands to an operation may be accessed directly from memory or they may need to be first loaded into temporary storage, depending on the class of instruction and choice of specific instruction.

Operand storage in the CPU	Where are operands kept other than in memory?
Number of explicit operands named per instruction	How many operands are named explicitly in a typical instruction?
Operand location	Can any ALU instruction operand be located in memory or must some or all of the operands be in internal storage in the CPU? If an operand is located in memory, how is the memory location specified?
Operations	What operations are provided in the instruction set?
Type and size of operands	What is the type and size of each operand and how is it specified?

FIGURE 3.1 A set of axes for alternative design choices in instruction sets. The type of storage provided for holding operands in the CPU, as opposed to in memory, is the major distinguishing factor among instruction set architectures. (All architectures known to the authors provide some temporary storage within the CPU.) The type of operand storage in the CPU sometimes dictates the number of operands explicitly named in an instruction. In one class of machines, the number of explicit operands may vary. Among recent instruction sets, the number of memory operands per instruction is another significant differentiating factor. The choice of what operations will be supported in instructions interacts less with other aspects of the architecture. Finally, specifying the data type and the size of an operand is largely independent of other instruction set choices.

G

Temporary storage provided	Examples	Explicit operands per ALU instruction	Destination for results	Procedure for accessing explicit operands
Stack	B5500, HP 3000/70	0	Stack	Push and pop onto or from the stack
Accumulator	PDP-8, Motorola 6809	1	Accumulator	Load/store accumulator
Register set	IBM 360, DEC VAX	2 or 3	Registers or memory	Load/store of registers, or memory

FIGURE 3.2 Some alternatives for storing operands within the CPU. Each alternative means that a different number of explicit operands is needed for an instruction with two source operands and a result operand. Instruction sets are usually classified by this internal state as stack machine, accumulator machine, or general-purpose register machine. While most architectures fit cleanly into one or another class, some architectures are hybrids of different approaches. The Intel 8086, for example, is halfway between a general-purpose register machine and an accumulator machine.

Figure 3.3 shows how the code sequence C = A + B would typically appear on these three classes of instruction sets. The primary advantages and disadvantages of each of these approaches are listed in Figure 3.4 (page 92).

While most early machines used stack or accumulator-style architectures, every machine designed in the past ten years and still surviving uses a general-purpose register architecture. The major reasons for the emergence of general-purpose register machines are twofold. First, registers—like other forms of

Stack	Accumulator	Register
PUSH A	LOAD A	LOAD R1,A
PUSH B	ADD B	ADD R1,B
ADD	STORE C	STORE C, R1
POP C		

FIGURE 3.3 The code sequence for C = A + B for three different instruction sets. It is assumed that A, B, and C all belong in memory and that the values of A and B cannot be destroyed.

Machine type	Advantages	Disadvantages
Stack	Simple model of expression evaluation (reverse polish). Short instructions can yield good code density.	A stack cannot be randomly accessed. This limitation makes it difficult to generate efficient code. It's also difficult to implement efficiently, since the stack becomes a bottleneck.
Accumulator	Minimizes internal state of machine. Short instructions.	Since accumulator is only temporary storage, memory traffic is highest for this approach.
Register	Most general model for code generation.	All operands must be named, leading to longer instructions.

FIGURE 3.4 Primary advantages and disadvantages of each class of machine. These advantages and disadvantages are related to three issues: How well the structure matches the needs of a compiler; how efficient the approach is from an implementation viewpoint; and what the effective code size is relative to other approaches.

storage internal to the CPU—are faster than memory. Second, registers are easier for a compiler to use and can be used more effectively than other forms of internal storage. Because general-purpose register machines so dominate instruction set architectures today—and it seems unlikely that this will change in the future—it is only these architectures that we will consider from this point on. Yet even with this limitation, there is a large number of design alternatives to consider. Some designers have proposed the extension of the register concept to allow additional buffering of multiple sets of registers in a stack-like fashion. This additional level of memory hierarchy is examined in Chapter 8.

3.3 Operand Storage in Memory: Classifying General-Purpose Register Machines

The key advantages of general-purpose register machines arise from effective use of the registers by a compiler, both in computing expression values and, more globally, in using registers to hold variables. Registers permit more flexible ordering in evaluating expressions than do either stacks or accumulators. For example, on a register machine the expression (A*B) – (C*D) – (E*F) may be evaluated by doing the multiplications in any order, which may be more efficient due to the location of the operands or because of pipelining concerns (see Chapter 6). But on a stack machine the expression must be evaluated left to right, unless special operations or swaps of stack positions are done.

More important, registers can be used to hold variables. When variables are allocated to registers, the memory traffic is reduced, the program is sped up (since registers are faster than memory), and the code density improves (since a register can be named with fewer bits than can a memory location). Compiler writers would prefer that all registers be equivalent and unreserved. Many machines compromise this desire—especially older machines with many dedicated registers—effectively decreasing the number of general-purpose registers.

9

If the number of truly general-purpose registers is too small, trying to allocate variables to registers will not be profitable. Instead, the compiler will reserve all the uncommitted registers for use in expression evaluation.

How many registers are sufficient? The answer of course depends on how they are used by the compiler. Most compilers reserve some registers for expression evaluation, use some for parameter passing, and allow the remainder to be allocated to hold variables. To understand how many registers are sufficient, we really need to examine what variables can be allocated to registers and the allocation algorithm used. We deal with these in our discussion of compilers in Section 3.7 and examine measurements of register usage in that section.

There are two major instruction set characteristics that divide general-purpose register, or *GPR*, architectures. Both characteristics concern the nature of operands for a typical arithmetic or logical instruction, or ALU instruction. The first concerns whether an ALU instruction has two or three operands. In the three-operand format, the instruction contains a result and two source operands. In the two-operand format, one of the operands is both a source and a destination for the operation. The second distinction among GPR architectures concerns how many of the operands may be memory addresses in ALU instructions. The number of memory operands supported by a **typical** ALU instruction may vary from none to three. All possible combinations of these two attributes are shown in Figure 3.5, with examples of machines. While there are seven possible combinations, three serve to classify nearly all existing machines: *register–register* (also called *load/store*), *register–memory*, and *memory–memory*.

The advantages and disadvantages of each of these alternatives are shown in Figure 3.6 (page 94). Of course, these advantages and disadvantages are not absolutes. A GPR machine with memory–memory operations can easily be subsetted by the compiler and used as a register–register machine. The

Number of memory addresses per typical ALU instruction	Maximum number of operands allowed for a typical ALU instruction	Examples
0	2	IBM RT-PC
	3	SPARC, MIPS, HP Precision Architecture
1	2	PDP-10, Motorola 68000, IBM 360
	3	Part of IBM 360 (RS instructions)
2	2	PDP-11, National 32x32, part of IBM 360 (SS instructions)
	3	
3	3	VAX (also has two operand formats)

FIGURE 3.5 Possible combinations of memory operands and total operands per ALU instruction with examples of machines. Machines with no memory reference per ALU instruction are called load/store or register–register machines. Instructions with multiple memory operands per typical ALU instruction are called register–memory or memory–memory, according to whether they have one or more than one memory operand.

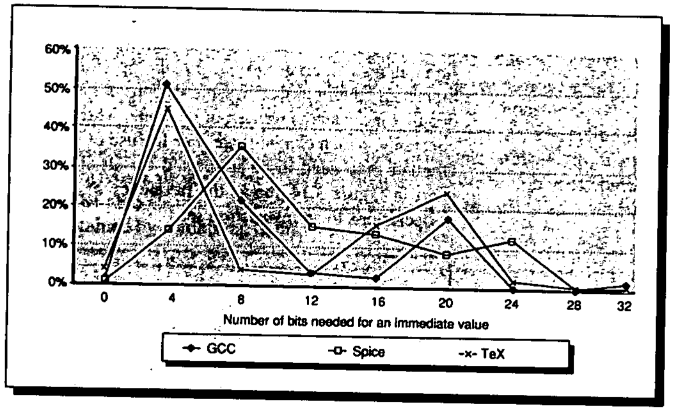


FIGURE 3.15 The distribution of immediate values is shown. The x axis shows the number of bits needed to represent the magnitude of an immediate value—0 means the immediate field value was 0. The vast majority of the immediate values are positive: Overall, less than 6% of the immediates are negative.These measurements were taken on a VAX, which supports a full range of immediates and sizes as operands to any instruction. The measured programs are the standard set—GCC, Spice, and TeX.

Encoding of Addressing Modes

How the addressing modes of operands are encoded depends on the range of addressing modes and the degree of independence between opcodes and modes. For a small number of addressing modes or opcode/addressing mode combinations, the addressing mode can be encoded in the opcode. This works for the IBM 360 with only five addressing modes and most operations offered in only one or two modes. For a large number of combinations, typically a separate *address specifier* is needed for each operand. The address specifier tells what addressing mode the operand is using. In Chapter 4, we will see how these two types of encodings are used in several real instruction formats.

When encoding the instructions, the number of registers and the number of addressing modes both have a significant impact on the size of instructions. This is because the addressing mode field and the register field may appear many times in a single instruction. In fact, for most instructions many more bits are consumed encoding addressing modes and register fields than in specifying the opcode. The architect must balance several competing forces when encoding the instruction set:

1. The desire to have as many registers and addressing modes as possible.

G

2. The impact of the size of the register and addressing mode fields on the average instruction size and hence on the average program size.
3. A desire to have instructions encode into lengths that will be easy to handle in the implementation. As a minimum, the architect wants instructions to be in multiples of bytes, rather than an arbitrary length. Many architects have chosen to use a fixed-length instruction to gain implementation benefits while sacrificing average code size.

Since the addressing modes and register fields make up such a large percentage of the instruction bits, their encoding will significantly affect how easy it is for an implementation to decode the instructions. The importance of having easily decoded instructions is discussed in Chapters 5 and 6.

3.5 Operations in the Instruction Set

The operators supported by most instruction set architectures can be categorized, as in Figure 3.16. In Section 3.8, we look at the use of operations in a general fashion (e.g. memory references, ALU operations, and branches). In Chapter 4, we will examine the use of various instruction operations in detail for four different architectures. Because the instructions used to implement control flow are largely independent of other instruction set choices and because the measurements of branch and jump behavior are also fairly independent of other measurements, we examine the use of control-flow instructions next.

Operator type	Examples
Arithmetic and logical	Integer arithmetic and logical operations: add, and, subtract, or
Data transfer	Loads/stores (move instructions on machines with memory addressing)
Control	Branch, jump, procedure call and return, traps
System	Operating system call, virtual memory management instructions
Floating point	Floating-point operations: add, multiply
Decimal	Decimal add, decimal multiply, decimal-to-character conversions
String	String move, string compare, string search

FIGURE 3.16 Categories of instruction operators and examples of each. All machines generally provide a full set of operations for the first three categories. The support for system functions in the instruction set varies widely among architectures, but all machines must have some instruction support for basic system functions. The amount of support in the instruction set for the last three categories may vary from none to an extensive set of special instructions. Floating-point instructions will be provided in any machine that is intended for use in an application that makes much use of floating point. These instructions are sometimes part of an optional instruction set. Decimal and string instructions are sometimes primitives, as in the VAX or the IBM 360, or may be synthesized by the compiler from simpler instructions. Examples of instruction sets appear in Appendix B, while Appendix C contains measurements of typical usage. We will examine four different instruction sets and their usage in detail in Chapter 4.

Instructions for Control Flow

As Figure 3.17 shows, there is no consistent terminology for instructions that change the flow of control. Until the IBM 7030, control-flow instructions were typically called *transfers*. Beginning with the 7030, the name *branch* began to be used. Later, machines introduced additional names. Throughout this book we will use *jump* when the change in control is unconditional and *branch* when the change is conditional.

Machine	Year	"Branch"	"Jump"
IBM 7030	1960	All control transfers—addressing is PC-relative	
IBM 360/370	1965	All control transfers—no PC-relative	
DEC PDP-11	1970	PC-relative only, conditional and unconditional	All addressing modes; unconditional only
Intel 8086	1978		All transfers are jumps; conditional jumps are PC-relative only
DEC VAX	1978	Same as PDP-11	Same as PDP-11
MIPS R2000	1986	Conditional control transfer, always PC-relative	Unconditional jumps and call instructions

FIGURE 3.17 Machines, dates, and the names associated with control transfers in their architectures. These names vary widely based on whether the transfer is conditional or unconditional and on whether it is PC-relative or not. The VAX, PDP-11, and MIPS R2000 architectures allow only PC-relative addressing for branches.

We can distinguish four different types of control-flow change:

1. Conditional branches
2. Jumps
3. Procedure calls
4. Procedure returns

We want to know the relative frequency of these events, as each event is different, may use different instructions, and may have different behavior. The frequencies of these control-flow instructions for a load/store machine running our benchmarks is shown in Figure 3.18.

The destination address of a branch must always be specified. This destination is specified explicitly in the instruction in the vast majority of cases—procedure return being the major exception—since the target is known at compile time. The most common way to specify the destination is to supply a displacement that is added to the *program counter*, or PC. Branches of this sort are called *PC-relative* branches. PC-relative branches are advantageous because the branch target is often near the current instruction, and specifying the position relative to the current PC requires fewer bits. Using PC-relative addressing also permits the code to run independent of where it is loaded. This property, called *position-independence*, can eliminate some work when the program is linked and is also useful in programs linked during execution.

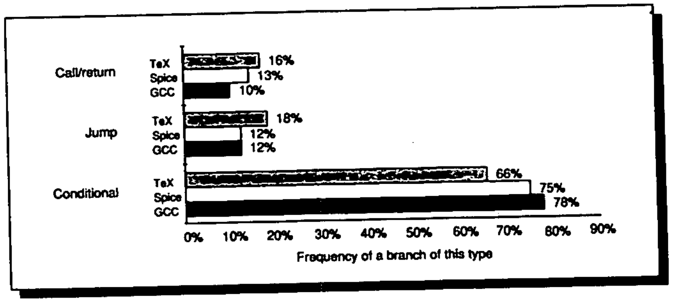


FIGURE 3.18 Breakdown of branches into three classes. Each branch is counted in one of three columns. Conditional branches clearly dominate. On average 90% of the jumps are PC-relative.

To implement returns and indirect branches in which the target is not known at compile time, a method other than PC-relative addressing is required. Here, there must be a way to specify the target dynamically, so that it can change at run-time. This may be as simple as naming a register that contains the target address. Alternatively, the branch may permit any addressing mode to be used to supply the target address.

A key question concerns how far branch targets are from branches. Knowing the distribution of these displacements will help in choosing what branch offsets to support and thus will affect the instruction length and encoding. Figure 3.19 (page 106) shows the distribution of displacements for PC-relative branches in instructions. About 75% of the branches are in the forward direction.

Since most changes in control flow are branches, deciding how to specify the branch condition is important. The three primary techniques in use and their advantages and disadvantages are shown in Figure 3.20 (page 106).

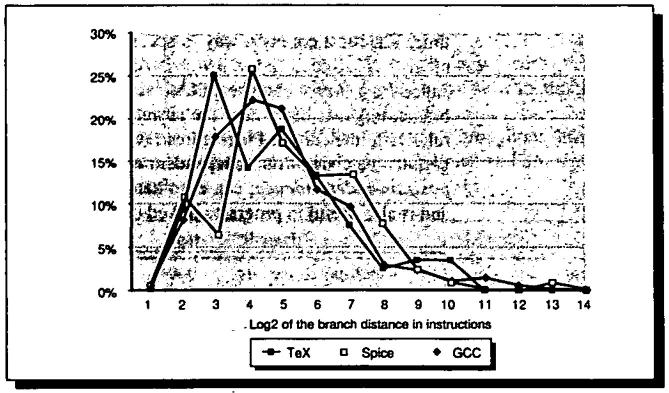


FIGURE 3.19 Branch distances in terms of number of instructions between the target and the branch instruction. The most frequent branches in Spice are to targets that are 8 to 15 instructions away (2^4). The weighted-arithmetic-mean branch target distance is 86 instructions (2^7). This tells us that short displacement fields often suffice for branches and that the designer can gain some encoding density by having a shorter instruction with a smaller branch displacement. These measurements were taken on a load/store machine (MIPS R2000 architecture). An architecture that requires fewer instructions for the same program, such as a VAX, would have shorter branch distances. Similarly, the number of bits needed for the displacement may change if the machine allows instructions to be arbitrarily aligned. A cumulative distribution of this branch displacement data is shown in Exercise 3.3 (see Figure 3.35 on page 133).

Name	How condition is tested	Advantages	Disadvantages
Condition code (CC)	Special bits are set by ALU operations, possibly under program control.	Sometimes condition is set for free.	CC is extra state. Condition codes constrain the ordering of instructions since they pass information from one instruction to a branch.
Condition register	Test arbitrary register with the result of a comparison.	Simple.	Uses up a register.
Compare and branch	Compare is part of the branch. Often compare is limited to subset.	One instruction rather than two for a branch.	May be too much work per instruction.

FIGURE 3.20 The major methods for evaluating branch conditions, their advantages, and disadvantages. Although condition codes can be set by ALU operations that are needed for other purposes, measurements on programs show that this rarely happens. The major implementation problems with condition codes arise when the condition code is set by a large or haphazardly chosen subset of the instructions, rather than being controlled by a bit in the instruction. Machines with compare and branch often limit the set of compares and use a condition register for more complex compares. Often, different techniques are used for branches based on floating-point comparison versus those based on integer comparison. This is reasonable since the number of branches that depend on floating-point comparisons is much smaller than the number depending on integer comparisons.

One of the most noticeable properties of branches is that a large number of the comparisons are simple equality or inequality tests, and a large number are comparisons with zero. Thus, some architectures choose to treat these comparisons as special cases, especially if a *compare and branch* instruction is being used. Figure 3.21 shows the frequency of different comparisons used for conditional branching. The data in Figure 3.14 said that a large percentage of the branches had an immediate operand (86%), and while not shown, 0 was the most heavily used immediate (83% of the immediates in branches). When we combine this with the data in Figure 3.21 we can see that a significant percentage (over 50%) of the compares in branches are simple tests for equality with zero.

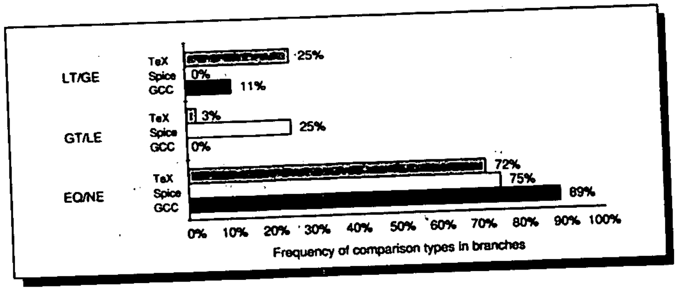


FIGURE 3.21 Frequency of different types of compares in conditional branches. This includes both the integer and floating-point compares in branches. Floating-point comparisons constitute 13% of the branch comparisons in Spice. Remember that earlier data in Figures 3.14 indicate that most comparisons are against an immediate operand. This immediate value is usually 0 (83% of the time).

Program	Percentage of backward branches	Percentage taken branches	Percentage of all control instructions that actually branch
GCC	26%	54%	63%
Spice	31%	51%	63%
TeX	17%	54%	70%
Average	25%	53%	65%

FIGURE 3.22 Branch direction, branch-taken frequency, and frequency that the PC is changed. The first column shows what percentage of all branches (both taken and untaken) are backward-going. The second column shows what percentage of all branches (remember that a branch is always conditional) are taken. The final column shows what percentage of all control-flow instructions actually cause a nonsequential transfer in the flow. This last column is computed by combining data from the second column and the data in Figure 3.18 (page 105).

We will say that a branch is *taken* if the condition tested by the branch is true and the next instruction to be executed is the target of the branch. All jumps, therefore, are taken. Figure 3.22 shows the branch-direction distribution, the frequency of taken (conditional) branches, and the percentage of control-flow instructions that change the PC. Most backward-going branches are loop branches, and typically loop branches are taken with about 90% probability.

Many programs have a higher percentage of loop branches, thus boosting the frequency of taken branches over 60%. Overall, branch behavior is application dependent and sometimes compiler dependent. Compiler dependencies arise because of changes to the control flow made by optimizing compilers to improve the execution time of loops.

Example

Assuming that 90% of the backward-going branches are taken, find the probability that a forward-going branch is taken using the averaged data in Figure 3.22.

Answer

The average frequency of taken branches is the sum of the backward-taken and forward-taken times their respective frequencies:

$$\begin{aligned}
\text{\% taken branches} &= (\text{\% taken backward} * \text{\% backward}) + \\
&\quad (\text{\% taken forward} * \text{\% forward}) \\
53\% &= (90\% * 25\%) + (\text{\% taken forward} * 75\%) \\
\text{\% taken forward} &= \frac{53\% - 22.5\%}{75\%} \\
\text{\% taken forward} &= 40.7\%
\end{aligned}$$

It is not unusual to see the majority of forward branches be untaken. The behavior of forward-going branches often varies among programs.

Procedure calls and returns include control transfer and possibly some state saving; at a minimum the return address must be saved somewhere. Some architectures provide a mechanism to save the registers, while others require the compiler to generate instructions. There are two basic conventions in use to save registers. *Caller-saving* means that the calling procedure must save the registers that it wants preserved for access after the call. *Callee-saving* means that the called procedure must save the registers it wants to use. There are times when caller save must be used due to access patterns to globally visible variables in two different procedures. For example, suppose we have a procedure P1 that calls procedure P2, and both procedures manipulate the global variable *x*. If P1 had allocated *x* to a register it must be sure to save *x* to a location known by P2 before the call to P2. A compiler's ability to discover when a called procedure

G

may access register-allocated quantities is complicated by the possibility of separate compilation, and situations where P2 may not touch *x*, but P2 can call another procedure, P3, that may access *x*. Because of these complications, most compilers will conservatively caller save any variable that may be accessed during a call.

In the cases where either convention could be used, some will be more optimal with callee-save and some will be more optimal with caller-save. As a result, the most sophisticated compilers use a combination of the two mechanisms, and the register allocator may choose which register to use for a variable based on the convention. Later in this chapter we will examine how well more sophisticated instructions match the needs of the compiler for this function, and in Chapter 8 we will look at hardware buffering schemes for supporting register save and restore.

3.6 Type and Size of Operands

How is the type of an operand designated? There are two primary alternatives: First, the type of an operand may be designated by encoding it in the opcode—this is the method used most often. Alternatively, the data can be annotated with tags that are interpreted by the hardware. These tags specify the type of the operand, and the operation is chosen accordingly. Machines with tagged data, however, are extremely rare. The Burroughs' architectures are the most extensive example of tagged architectures. Symbolics also built a series of machines that used tag data items for implementing LISP.

Usually the type of an operand—for example, integer, single-precision floating point, character—effectively gives its size. Common operand types include character (one byte), halfword (16 bits), word (32 bits), single-precision floating point (also one word), and double-precision floating point (two words). Characters are represented as either EBCDIC, used by the IBM mainframe architectures, or ASCII, used by everyone else. Integers are almost universally represented as two's complement binary numbers. Until recently, most computer manufacturers chose their own floating-point representation. However, in the past few years, a standard for floating point, the IEEE standard 754, has become the choice of most new computers. The IEEE floating-point standard is discussed in detail in Appendix A.

Some architectures provide operations on character strings, although such operations are usually quite limited and treat each byte in the string as a single character. Typical operations supported on character strings are comparisons and moves.

For business applications, some architectures support a decimal format, usually called *packed decimal*. Packed decimal is *binary-coded decimal*—four bits are used to encode the values 0–9, and two decimal digits are packed into each byte. Numeric character strings are sometimes called unpacked decimal, and stalls. A *stall* is where an instruction must pause one or more clock cycles waiting for some resource to be available. In this chapter stalls occur only when waiting for memory; in the next chapter we'll see other reasons for stalls.

Many machines approach this problem by having the hardware stall a microinstruction that tries to access the memory-data register before the memory operation is completed. (This can be accomplished by freezing the microinstruction address so that the same microinstruction is executed until the condition is met.) The instant the memory reference is ready, the microinstruction that needs the data is allowed to complete, avoiding the extra clock delay to access control memory.

Reducing CPI by Parallelism

Sometimes CPI can be reduced with more operations per microinstruction. This technique, which usually requires a wider microinstruction, increases parallelism with more datapath operations. It is another characteristic of machines labeled horizontal. Examples of this performance gain can be seen in the fact that the fastest models of each family in Figure 5.8 also have the widest microinstructions. Making the microinstruction wider does not guarantee increased performance, however. An example where the potential gain was not realized is found in a microprocessor very similar to the 8086, except that another bus was added to the datapath, requiring six more bits in its microinstruction. This could have reduced the execution phase from three clock cycles to two for many popular 8086 instructions. Unfortunately, these popular macroinstructions were grouped with macroinstructions that couldn't take advantage of this optimization, so they all had to run at the slower rate.

5.6 Interrupts and Other Entanglements

Control is the hard part of processor design, and the hard part of control is *interrupts*—events other than branches that change the normal flow of instruction execution. Detecting interrupt conditions within an instruction can often be on the critical timing path of a machine, possibly affecting the clock cycle time, and thus performance. Without proper attention to interrupts during design, adding interrupts to a complicated implementation can even foul up the works so as to make the design impracticable.

Invented to detect arithmetic errors and signal real-time events, interrupts have been handed a flock of difficult duties. Here are 11 examples:

I/O device request

Invoking an operating system service from a user program

Tracing instruction execution

Breakpoint (programmer-requested interrupt)

Arithmetic overflow or underflow

Page fault (not in main memory)

Misaligned memory accesses (if alignment is required)

Memory-protection violation

Using an undefined instruction

Hardware malfunctions

Power failure

	IBM 360	VAX	Motorola 680x0	Intel 80x86
I/O device request	Input/output interruption	Device interrupt	Exception (Level 0...7 autovector)	Vectored interrupt
Invoking the operating system service from a user program	Supervisor call interruption	Exception (change mode supervisor trap)	Exception (unimplemented instruction)—on Macintosh	Interrupt (INT instruction)
Tracing instruction execution	NA	Exception (trace fault)	Exception (trace)	Interrupt (single-step trap)
Breakpoint	NA	Exception (breakpoint fault)	Exception (illegal instruction or breakpoint)	Interrupt (breakpoint trap)
Arithmetic overflow or underflow	Program interruption (overflow or underflow exception)	Exception (integer overflow trap or floating underflow fault)	Exception (floating-point coprocessor errors)	Interrupt (overflow trap or math unit exception)
Page fault (not in main memory)	NA (only in 370)	Exception (translation not valid fault)	Exception (memory-management unit errors)	Interrupt (page fault)
Misaligned memory accesses	Program interruption (specification exception)	NA	Exception (address error)	NA
Memory protection violations	Program interruption (protection exception)	Exception (access control violation fault)	Exception (bus error)	Interrupt (protection exception)
Using undefined instructions	Program interruption (operation exception)	Exception (opcode privileged/ reserved fault)	Exception (illegal instruction or break-point/unimplemented instruction)	Interrupt (invalid opcode)
Hardware malfunctions	Machine-check interruption	Exception (machine-check abort)	Exception (bus error)	NA
Power failure	Machine-check interruption	Urgent interrupt	NA	Nonmaskable interrupt

FIGURE 5.9 Names of 11 interrupt classes on four computers. Every event on the IBM 360 and 80x86 is called an *interrupt*, while every event on the 680x0 is called an *exception*. VAX divides events into *interrupts* or *exceptions*. Adjectives *device*, *software*, and *urgent* are used with VAX interrupts, while VAX exceptions are subdivided into *faults*, *traps*, and *aborts*.

The enlarged responsibility of interrupts has led to the confusing situation of each computer vendor inventing a different term for the same event, as Figure 5.9 on page 215 illustrates. Intel and IBM still call such events *interrupts*, but Motorola calls them *exceptions;* and, depending on the circumstances, DEC calls them *exceptions, faults, aborts, traps, or interrupts.* To give some idea of how often interrupts occur, Figure 5.10 shows the frequency on the VAX 8800.

Event	Time between events
I/O interrupt	2.7 ms
Interval timer interrupt	10.0 ms
Software interrupt	1.5 ms
Any interrupt	0.9 ms
Any hardware interrupt	2.1 ms

FIGURE 5.10 Frequency of different interrupts on the VAX 8800 running a multiuser workload on the VMS timesharing system. Real-time operating systems used in embedded controllers may have a higher interrupt rate than a general-purpose timesharing system. (Collected by Clark, Bannon, and Keller [1988].)

Clearly, there is no consistent convention for naming these events. Rather than imposing one, then, let's review the reasons for the different names. The events can be characterized on five independent axes:

1. Synchronous versus asynchronous. If the event occurs at the same place every time the program is executed with the same data and memory allocation, the event is synchronous. With the exception of hardware malfunctions, asynchronous events are caused by devices external to the processor and memory.
2. User request versus coerced. If the user task directly asks for it, it is a user-request event.
3. User maskable versus user nonmaskable. If it can be masked or disabled by a user task, the event is user maskable.
4. Within versus between instructions. This classification depends on whether the event prevents instruction completion by occurring in the middle of execution—no matter how short—or whether it is recognized between instructions.
5. Resume versus terminate. If the program's execution stops after the interrupt, it is a terminating event.

G

The difficult task is implementing interrupts occurring within instructions where the instruction must be resumed. Another program must be invoked to collect the state of the program, correct the cause of an interrupt, and then restore the state of the program before an instruction can be tried again.

Figure 5.11 classifies the examples from Figure 5.9 according to these five categories.

	Synchronous vs. asynchronous	User request vs. coerced	User maskable vs. nonmaskable	Within vs. between instructions	Resume vs. terminate
I/O device request	Asynchronous	Coerced	Nonmaskable	Between	Resume
Invoking operating system service	Synchronous	User request	Nonmaskable	Between	Resume
Tracing instruction execution	Synchronous	User request	User maskable	Between	Resume
Breakpoint	Synchronous	User request	User maskable	Between	Resume
Integer arithmetic overflow	Synchronous	Coerced	User maskable	Within	Terminate
Floating-point arithmetic overflow or underflow	Synchronous	Coerced	User maskable	Within	Resume
Page fault	Synchronous	Coerced	Nonmaskable	Within	Resume
Misaligned memory accesses	Synchronous	Coerced	User maskable	Within	Terminate
Memory-protection violations	Synchronous	Coerced	Nonmaskable	Within	Terminate
Using undefined instructions	Synchronous	Coerced	Nonmaskable	Within	Terminate
Hardware malfunctions	Asynchronous	Coerced	Nonmaskable	Within	Terminate
Power failure	Asynchronous	Coerced	Nonmaskable	Within	Terminate

FIGURE 5.11 The events of Figure 5.9 classified using five categories.

How Control Checks for Interrupts

Integrating interrupts with control means modifying the finite-state diagram to check for interrupts. Interrupts that occur between instructions are checked either at the beginning of the finite-state diagram—before an instruction is decoded—or at the end—after the execution of an instruction is completed. Interrupts that can occur within an instruction are generally detected in the state that causes the action or in a state that follows it. For example, Figure 5.12 shows Figure 5.4 (page 207) modified to check for interrupts.

We assume DLX transfers the return address into a new programmer-visible register, the interrupt return-address register. Control then loads PC with the address of the interrupt routine for that interrupt.

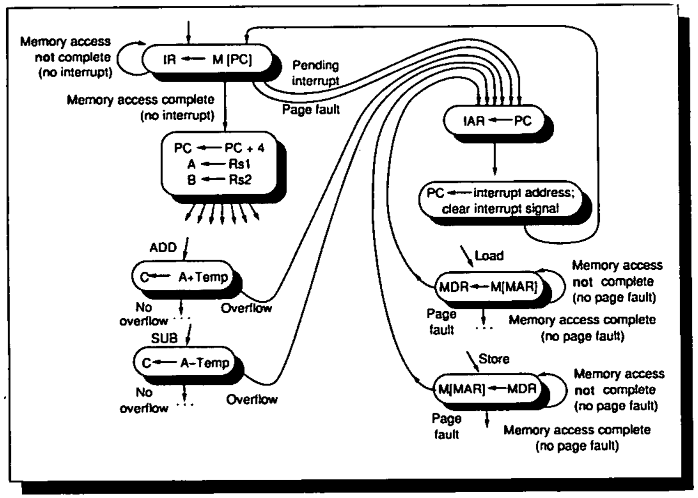

Memory access not complete (no interrupt)
IR ← M [PC]
Pending interrupt
Memory access complete (no interrupt)
Page fault
PC ← PC + 4
A ← Rs1
B ← Rs2
ADD
C ← A+Temp
No overflow
Overflow
SUB
C ← A−Temp
No overflow
Overflow
IAR ← PC
PC ← interrupt address; clear interrupt signal
Load
MDR ← M[MAR]
Memory access not complete (no page fault)
Page fault
Memory access complete (no page fault)
Store
M[MAR] ← MDR
Memory access not complete (no page fault)
Page fault
Memory access complete (no page fault)

FIGURE 5.12 The top-level view of the DLX finite-state diagram (Figure 5.4 on page 207) modified to check for interrupts. Either a between interrupt or an instruction page fault invokes the control that saves the PC and then loads it with the address of the appropriate interrupt routine. The lower portion of the figure shows interrupts resulting in page faults of data accesses or arithmetic overflow.

What's Hard About Interrupts

The conflicting terminology is confusing, but that is not what makes the hard part of control hard. Even though interrupts are rare, the hardware must be designed so that the full state of the machine can be saved, including an indication of the offending event, and the PC of the instruction to be executed after the interrupt is serviced. This difficulty is exacerbated by events occurring during the middle of execution, for many instructions also require the hardware to restore the machine to the state just before the event occurred—the beginning of the instruction. This last requirement is so difficult that computers are awarded the title *restartable* if they pass that test. That supercomputers and many early microprocessors do not earn that badge of honor illustrates both the difficulty of interrupts and the potential cost in hardware complexity and execution speed.

No engineers deserve more admiration than those who built the first VAX, DEC's first restartable computer. The variable-length instructions mean the computer can fetch 50 bytes of one instruction before discovering that the next byte of the instruction is not in main memory—a situation that requires the saved PC to point 50 bytes earlier. Imagine the difficulties of restarting an instruction with six operands, each of which could be misaligned and thus be partially in memory and partially on disk!

9

The instructions that are hardest to restart are those that modify some of the machine state before it is known whether interrupts can occur. The VAX autoincrement and autodecrement addressing modes would naturally modify registers during the addressing phase of execution rather than at the writeback phase, and so would be vulnerable to this difficulty. To avoid this problem, recent VAXes keep a history queue of the register specifiers and the operations on the registers, so that the operations can be reversed on an interrupt. Another approach, used on the earlier VAXes, is to record the specifiers and the original values of the registers, restoring the original values on interrupt. (The primary difference is that it only takes a few bits to record how the address was changed due to autoincrement or autodecrement versus the full 32-bit register value.)

It is not just addressing modes that make the VAX difficult to restart; long-running instructions mean that interrupts must be checked in the middle of execution to prevent long interrupt latency. MOVC3, for example, copies up to 2^{16} bytes and can take tens of milliseconds to finish—far too long to wait for an urgent event. On the other hand, even if there were a way to undo copying in the middle of execution so that MOVC3 could be restarted, interrupts would occur so frequently, relative to this long-running instruction (see Figure 5.10 on page 216), that MOVC3 would be restarted repeatedly under those conditions. Such wasted effort from incomplete copies would render MOVC3 worse than useless.

DEC divided the problem to conquer it. First, the operands—source address, length, and destination address—are fetched from memory and placed into general-purpose registers R1, R2, and R3. If an interrupt occurs during this first phase, these registers are restored, and the MOVC3 is restarted from scratch. After this first phase, every time a byte is copied, the length (R2) is decremented and addresses (R1 and R3) are incremented. If an interrupt occurs during this second phase, MOVC3 sets the *first part done* (FPD) bit in the program status word. When the interrupt is serviced and the instruction is reexecuted, it first checks the FPD bit to see if the operands have already been placed in registers. If so, the VAX doesn't fetch the address and length operands, but just continues with the current values in the registers, since that is all that remains to be copied. This permits more rapid response to interrupts while allowing long-running instructions to make progress between interrupts.

IBM had a similar problem. The 360 included the MVC instruction, which copies up to 256 bytes of data. For the early machines without virtual memory, the machine simply waited until the instruction was completed before servicing interrupts. With the inclusion of virtual memory in the 370, the problem could no longer be ignored. Control first tries to access all possible pages, forcing all possible virtual memory miss interrupts to occur before moving any data. If any interrupts occur in this phase, the instruction is restarted. Control then ignores interrupts until the instruction is complete. To allow longer copies, the 370 includes MOVCL, which can move up to 2^{24} bytes. The operands are in registers and are updated as a part of execution—like the VAX, except that there is no need for FPD since the operands are always in registers. (Or, to speak historically, the VAX solution is like the IBM 370, which came first.)

5.7 Putting It All Together: Control for DLX

The control for DLX is presented here to tie together the ideas from the previous three sections. We begin with a finite-state diagram to represent hardwired control and end with microprogrammed control. Both versions of DLX control are used to demonstrate tradeoffs to reduce cost or to improve performance. Because the figures are already too large, the checking for data page faults or arithmetic overflow shown in Figure 5.12 (page 218) is not included in this section. (Exercise 5.12 adds them.)

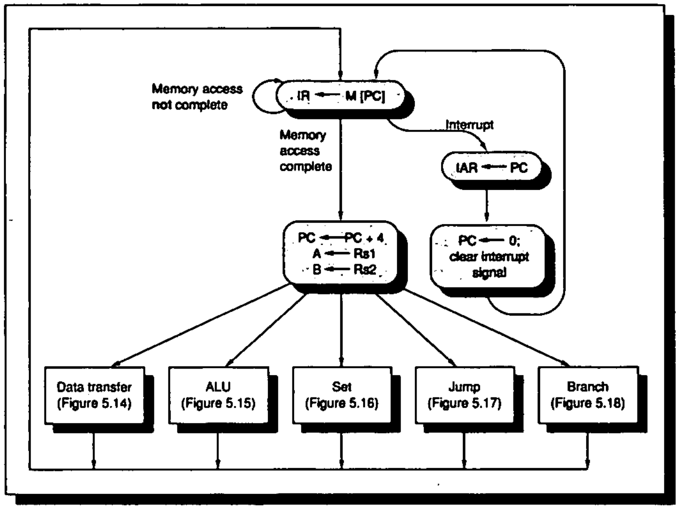


FIGURE 5.13 The top-level view of the DLX finite-state diagram for the non-floating-point instructions. The first two steps of instruction execution—instruction fetch and instruction decode/register fetch—are shown. The first state repeats until the instruction is fetched from memory or an interrupt is detected. If an interrupt is detected, the PC is saved in IAR and PC is set to the address of the interrupt routine. The last three steps of instruction execution—execution/effective address, memory access, and write back—are shown in Figures 5.14 to 5.18 on pages 221–224.

9

Rather than trying to draw the DLX finite-state machine in a single figure showing all 52 states, Figure 5.13 (see page 220) shows just the top level, containing 4 states plus references to the rest of the states detailed in Figures 5.14 (below) through 5.18 (page 224). Unlike Figure 5.2 (page 205), Figure 5.13 takes advantage of the change to the datapath allowing PC to address memory directly without going through MAR (Figure 5.4 on page 207).

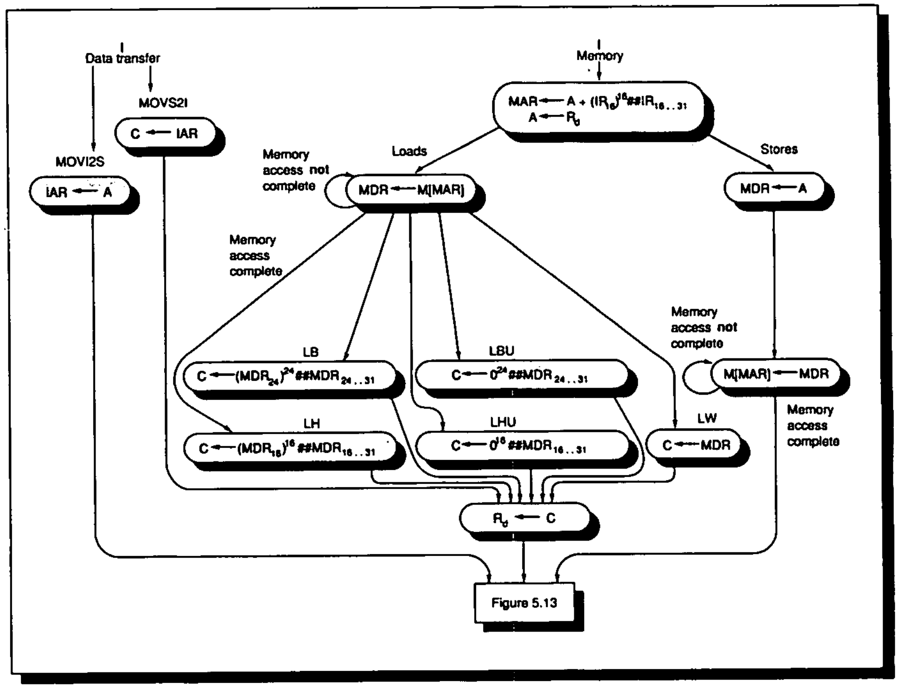


FIGURE 5.14 The effective address calculation, memory-access, and write-back states for the memory-access and data-transfer instructions of DLX. For loads, the second state repeats until the data is fetched from memory. The final state of stores repeats until the write is complete. While the operation of all five loads is shown in the states of this figure, the proper operation of writes depends on the memory system writing bytes and halfwords, without disturbing the rest of the word in memory, and correctly aligning the bytes and halfwords (see Figure 3.10, page 97) over the proper bytes of memory. On completion of execution control transfers to Figure 5.13, found on page 220.

intel®

Intel Architecture Software Developer's Manual

Volume 1: Basic Architecture

NOTE: The *Intel Architecture Software Developer's Manual* consists of three books: *Basic Architecture*, Order Number 243190; *Instruction Set Reference Manual*, Order Number 243191; and the *System Programming Guide*, Order Number 243192.
Please refer to all three volumes when evaluating your design needs.

1997

Information in this document is provided in connection with Intel products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

Intel's Intel Architecture processors (e.g., Pentium® processor, Pentium processor with MMX™ technology, Pentium Pro processor, and Pentium II processor) may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an ordering number and are referenced in this document, or other Intel literature, may be obtained from:

Intel Corporation
P.O. Box 7641
Mt. Prospect IL 60056-7641

or call 1-800-879-4683
or visit Intel's website at http:\\www.intel.com

* Third-party brands and names are the property of their respective owners.

G

Although all of these registers are available for general storage of operands, results, and pointers, caution should be used when referencing the ESP register. The ESP register holds the stack pointer and as a general rule should not be used for any other purpose.

Many instructions assign specific registers to hold operands. For example, string instructions use the contents of the ECX, ESI, and EDI registers as operands. When using a segmented memory model, some instructions assume that pointers in certain registers are relative to specific segments. For instance, some instructions assume that a pointer in the EBX register points to a memory location in the DS segment.

The special uses of general-purpose registers by instructions are described in Chapter 6, *Instruction Set Summary*, in this volume and Chapter 3, *Instruction Set Reference*, in the *Intel Architecture Software Developer's Manual, Volume 2*. The following is a summary of these special uses:

- EAX—Accumulator for operands and results data.
- EBX—Pointer to data in the DS segment.

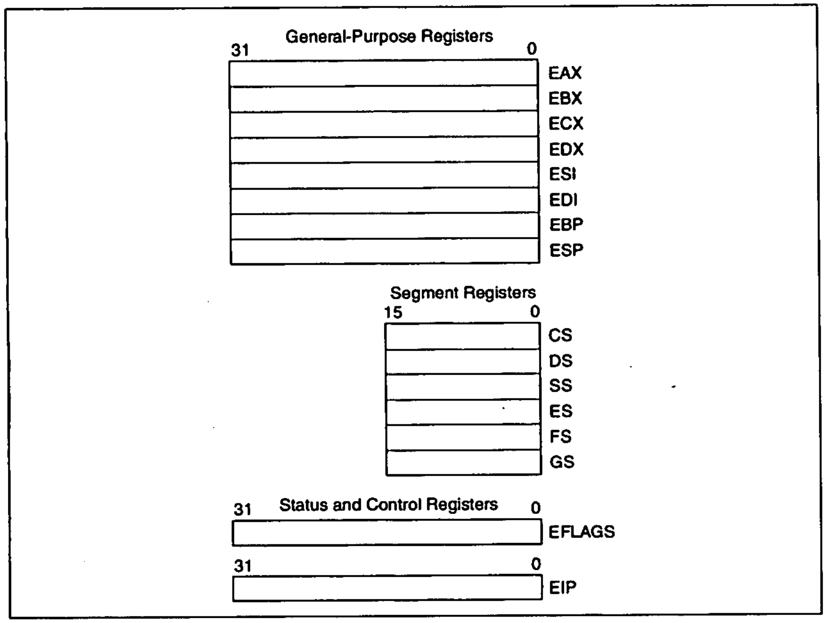


Figure 3-3. Application Programming Registers

- ECX—Counter for string and loop operations.
- EDX—I/O pointer.
- ESI—Pointer to data in the segment pointed to by the DS register; source pointer for string operations.
- EDI—Pointer to data (or destination) in the segment pointed to by the ES register; destination pointer for string operations.
- ESP—Stack pointer (in the SS segment).
- EBP—Pointer to data on the stack (in the SS segment).

As shown in Figure 3-4, the lower 16 bits of the general-purpose registers map directly to the register set found in the 8086 and Intel 286 processors and can be referenced with the names AX, BX, CX, DX, BP, SP, SI, and DI. Each of the lower two bytes of the EAX, EBX, ECX, and EDX registers can be referenced by the names AH, BH, CH, and DH (high bytes) and AL, BL, CL, and DL (low bytes).

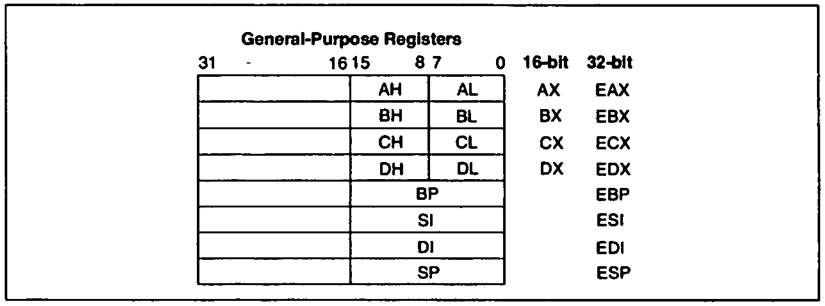


Figure 3-4. Alternate General-Purpose Register Names

3.6.2. Segment Registers

The segment registers (CS, DS, SS, ES, FS, and GS) hold 16-bit segment selectors. A segment selector is a special pointer that identifies a segment in memory. To access a particular segment in memory, the segment selector for that segment must be present in the appropriate segment register.

When writing application code, you generally create segment selectors with assembler directives and symbols. The assembler and other tools then create the actual segment selector values associated with these directives and symbols. If you are writing system code, you may need to create segment selectors directly. (A detailed description of the segment-selector data structure is given in Chapter 3, *Protected-Mode Memory Management*, of the *Intel Architecture Software Developer's Manual, Volume 3*.)

How segment registers are used depends on the type of memory management model that the operating system or executive is using. When using the flat (unsegmented) memory model, the segment registers are loaded with segment selectors that point to overlapping segments, each of which begins at address 0 of the linear address space (as shown in Figure 3-5). These overlapping segments then comprise the linear-address space for the program. (Typically, two overlapping segments are defined: one for code and another for data and stacks. The CS segment register points to the code segment and all the other segment registers point to the data and stack segment.)

When using the segmented memory model, each segment register is ordinarily loaded with a different segment selector so that each segment register points to a different segment within the linear-address space (as shown in Figure 3-6). At any time, a program can thus access up to six segments in the linear-address space. To access a segment not pointed to by one of the segment registers, a program must first load the segment selector for the segment to be accessed into a segment register.

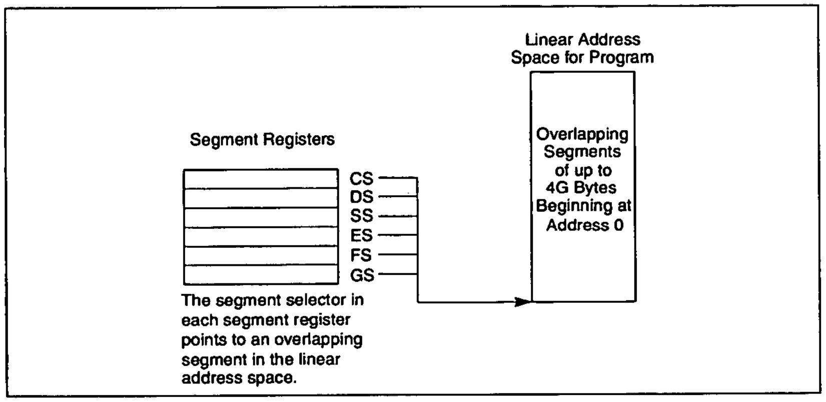

Linear Address Space for Program
Segment Registers
CS
DS
SS
ES
FS
GS
Overlapping Segments of up to 4G Bytes Beginning at Address 0
The segment selector in each segment register points to an overlapping segment in the linear address space.

Figure 3-5. Use of Segment Registers for Flat Memory Model

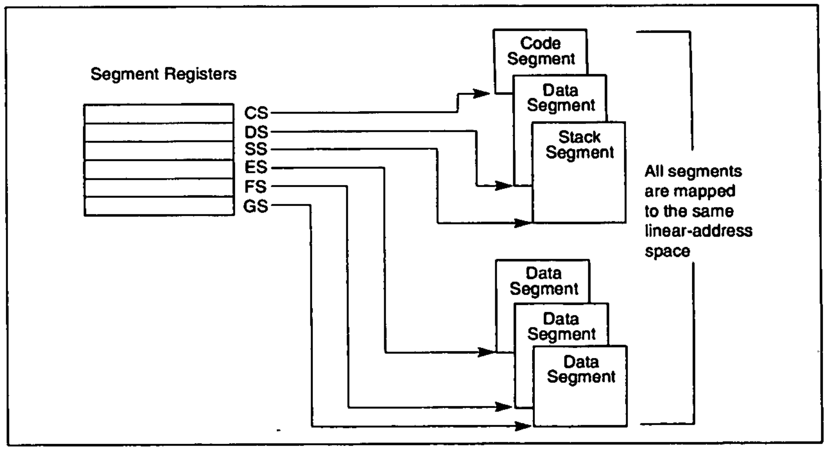


Figure 3-6. Use of Segment Registers In Segmented Memory Model

Each of the segment registers is associated with one of three types of storage: code, data, or stack). For example, the CS register contains the segment selector for the **code segment**, where the instructions being executed are stored. The processor fetches instructions from the code segment, using a logical address that consists of the segment selector in the CS register and the contents of the EIP register. The EIP register contains the linear address within the code segment of the next instruction to be executed. The CS register cannot be loaded explicitly by an application program. Instead, it is loaded implicitly by instructions or internal processor operations that change program control (such as, procedure calls, interrupt handling, or task switching).

The DS, ES, FS, and GS registers point to four **data segments**. The availability of four data segments permits efficient and secure access to different types of data structures. For example, four separate data segments might be created: one for the data structures of the current module, another for the data exported from a higher-level module, a third for a dynamically created data structure, and a fourth for data shared with another program. To access additional data segments, the application program must load segment selectors for these segments into the DS, ES, FS, and GS registers, as needed.

The SS register contains the segment selector for a **stack segment**, where the procedure stack is stored for the program, task, or handler currently being executed. All stack operations use the SS register to find the stack segment. Unlike the CS register, the SS register can be loaded explicitly, which permits application programs to set up multiple stacks and switch among them.

See Section 3.3., "Memory Organization", for an overview of how the segment registers are used in real-address mode.

9

The four segment registers CS, DS, SS, and ES are the same as the segment registers found in the Intel 8086 and Intel 286 processors and the FS and GS registers were introduced into the Intel Architecture with the Intel386 family of processors.

3.6.3. EFLAGS Register

The 32-bit EFLAGS register contains a group of status flags, a control flag, and a group of system flags. Figure 3-7 defines the flags within this register. Following initialization of the processor (either by asserting the RESET pin or the INIT pin), the state of the EFLAGS register is 00000002H. Bits 1, 3, 5, 15, and 22 through 31 of this register are reserved. Software should not use or depend on the states of any of these bits.

Some of the flags in the EFLAGS register can be modified directly, using special-purpose instructions (described in the following sections). There are no instructions that allow the whole register to be examined or modified directly. However, the following instructions can be used to move groups of flags to and from the procedure stack or the EAX register: LAHF, SAHF, PUSHF, PUSHFD, POPF, and POPFD. After the contents of the EFLAGS register have been transferred to the procedure stack or EAX register, the flags can be examined and modified using the processor's bit manipulation instructions (BT, BTS, BTR, and BTC).

When suspending a task (using the processor's multitasking facilities), the processor automatically saves the state of the EFLAGS register in the task state segment (TSS) for the task being suspended. When binding itself to a new task, the processor loads the EFLAGS register with data from the new task's TSS.

When a call is made to an interrupt or exception handler procedure, the processor automatically saves the state of the EFLAGS registers on the procedure stack. When an interrupt or exception is handled with a task switch, the state of the EFLAGS register is saved in the TSS for the task being suspended.

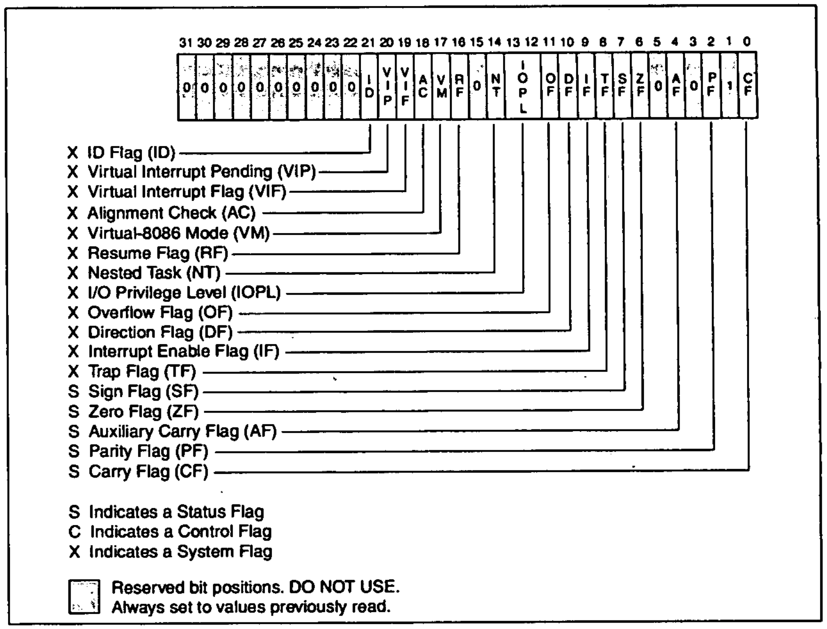

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 ID VIP VIF AC VM RF 0 NT IOPL OF DF IF TF SF ZF 0 AF 0 PF 1 CF
X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
X Overflow Flag (OF)
X Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)
S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag
Reserved bit positions. DO NOT USE.
Always set to values previously read.

Figure 3-7. EFLAGS Register

As the Intel Architecture has evolved, flags have been added to the EFLAGS register, but the function and placement of existing flags have remained the same from one family of the Intel Architecture processors to the next. As a result, code that accesses or modifies these flags for one family of Intel Architecture processors works as expected when run on later families of processors.

3.6.3.1. STATUS FLAGS

The status flags (bits 0, 2, 4, 6, 7, and 11) of the EFLAGS register indicate the results of arithmetic instructions, such as the ADD, SUB, MUL, and DIV instructions. The functions of the status flags are as follows:

CF (bit 0) **Carry flag.** Set if an arithmetic operation generates a carry or a borrow out of the most-significant bit of the result; cleared otherwise. This flag indicates an overflow condition for unsigned-integer arithmetic. It is also used in multiple-precision arithmetic.

7. Loads the segment selector for the new code segment and the new instruction pointer from the call gate into the CS and EIP registers, respectively.
8. Begins execution of the called procedure at the new privilege level.

When executing a return from the privileged procedure, the processor performs these actions:

1. Performs a privilege check.
2. Restores the CS and EIP registers to their values prior to the call.
3. (If the RET instruction has an optional *n* argument.) Increments the stack pointer by the number of bytes specified with the *n* operand to release parameters from the stack. If the call gate descriptor specifies that one or more parameters be copied from one stack to the other, a RET *n* instruction must be used to release the parameters from both stacks. Here, the *n* operand specifies the number of bytes occupied on each stack by the parameters. On a return, the processor increments ESP by *n* for each stack to step over (effectively remove) these parameters from the stacks.
4. Restores the SS and ESP registers to their values prior to the call, which causes a switch back to the stack of the calling procedure.
5. (If the RET instruction has an optional *n* argument.) Increments the stack pointer by the number of bytes specified with the *n* operand to release parameters from the stack (see explanation in step 3).
6. Resumes execution of the calling procedure.

See Chapter 4, *Protection*, in the *Intel Architecture Software Developer's Manual, Volume 3*, for detailed information on calls to privileged levels and the call gate descriptor.

4.4. INTERRUPTS AND EXCEPTIONS

The processor provides two mechanisms for interrupting program execution: interrupts and exceptions:

- An **interrupt** is an asynchronous event that is typically triggered by an I/O device.
- An **exception** is a synchronous event that is generated when the processor detects one or more predefined conditions while executing an instruction. The Intel architecture specifies three classes of exceptions: faults, traps, and aborts.

The processor responds to interrupts and exceptions in essentially the same way. When an interrupt or exception is signaled, the processor halts execution of the current program or task and switches to a handler procedure that has been written specifically to handle the interrupt or exception condition. The processor accesses the handler procedure through an entry in the interrupt descriptor table (IDT). When the handler has completed handling the interrupt or exception, program control is returned to the interrupted program or task.

The operating system, executive, and/or device drivers normally handle interrupts and exceptions independently from application programs or tasks. Application programs can, however, access the interrupt and exception handlers incorporated in an operating system or executive through assembly-language calls. The remainder of this section gives a brief overview of the processor's interrupt and exception handling mechanism. See Chapter 5, *Interrupt and Exception Handling* in the *Intel Architecture Software Developer's Manual, Volume 3*, for a detailed description of this mechanism.

The Intel Architecture defines 16 predefined interrupts and exceptions and 224 user defined interrupts, which are associated with entries in the IDT. Each interrupt and exception in the IDT is identified with a number, called a **vector**. Table 4-1 lists the interrupts and exceptions with entries in the IDT and their respective vector numbers. Vectors 0 through 8, 10 through 14, and 16 through 18 are the predefined interrupts and exceptions, and vectors 32 through 255 are the user-defined interrupts, called **maskable interrupts**.

Note that the processor defines several additional interrupts that do not point to entries in the IDT; the most notable of these interrupts is the SMI interrupt. See "Exception and Interrupt Vectors" in Chapter 5 of the *Intel Architecture Software Developer's Manual, Volume 3*, for more information about the interrupts and exceptions that the Intel Architecture supports.

When the processor detects an interrupt or exception, it does one of the following things:

- Executes an implicit call to a handler procedure.
- Executes an implicit call to a handler task.

4.4.1. Call and Return Operation for Interrupt or Exception Handling Procedures

A call to an interrupt or exception handler procedure is similar to a procedure call to another protection level (see Section 4.3.6., "CALL and RET Operation Between Privilege Levels"). Here, the interrupt vector references one of two kinds of gates: an **interrupt gate** or a **trap gate**. Interrupt and trap gates are similar to call gates in that they provide the following information:

- Access rights information.
- The segment selector for the code segment that contains the handler procedure.
- An offset into the code segment to the first instruction of the handler procedure.

The difference between an interrupt gate and a trap gate is as follows. If an interrupt or exception handler is called through an interrupt gate, the processor clears the interrupt enable (IF) flag in the EFLAGS register to prevent subsequent interrupts from interfering with the execution of the handler. When a handler is called through a trap gate, the state of the IF flag is not changed.

If the code segment for the handler procedure has the same privilege level as the currently executing program or task, the handler procedure uses the current stack; if the handler executes at a more privileged level, the processor switches to the stack for the handler's privilege level.

4

Table 4-1. Exceptions and Interrupts

Vector No.	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
9		CoProcessor Segment Overrun (reserved)	Floating-point instruction.[2]
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		(Intel reserved. Do not use.)	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory.[3]
18	#MC	Machine Check	Error codes (if any) and source are model dependent.[4]
19-31		(Intel reserved. Do not use.)	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT *n* instruction.

1. The UD2 instruction was introduced in the Pentium® Pro processor.
2. Intel Architecture processors after the Intel386™ processor do not generate this exception.
3. This exception was introduced in the Intel486™ processor.
4. This exception was introduced in the Pentium processor and enhanced in the Pentium Pro processor.

9

If no stack switch occurs, the processor does the following when calling an interrupt or exception handler (see Figure 4-5):

1. Pushes the current contents of the EFLAGS, CS, and EIP registers (in that order) on the stack.
2. Pushes an error code (if appropriate) on the stack.
3. Loads the segment selector for the new code segment and the new instruction pointer (from the interrupt gate or trap gate) into the CS and EIP registers, respectively.
4. If the call is through an interrupt gate, clears the IF flag in the EFLAGS register.
5. Begins execution of the handler procedure at the new privilege level.

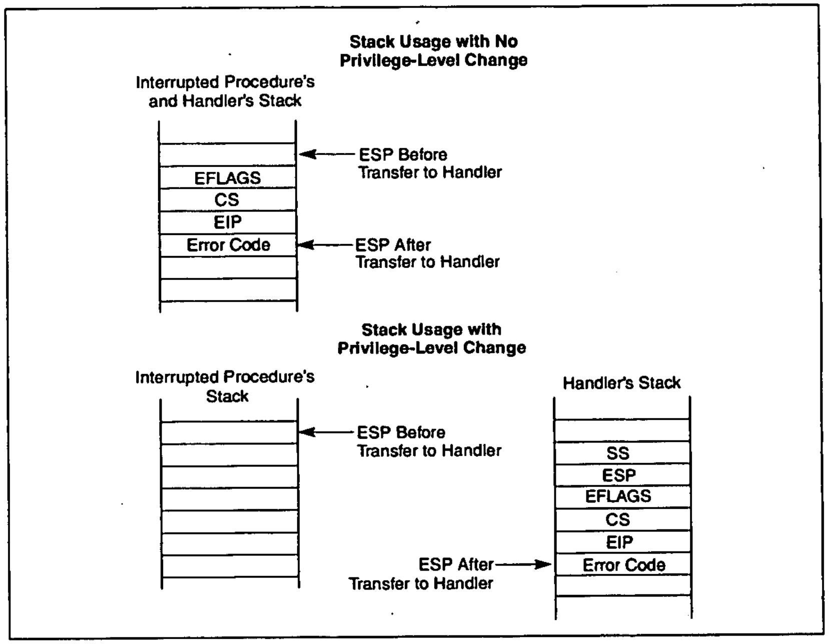

Stack Usage with No Privilege-Level Change
Interrupted Procedure's and Handler's Stack
EFLAGS
CS
EIP
Error Code
ESP Before Transfer to Handler
ESP After Transfer to Handler
Stack Usage with Privilege-Level Change
Interrupted Procedure's Stack
ESP Before Transfer to Handler
Handler's Stack
SS
ESP
EFLAGS
CS
EIP
Error Code
ESP After Transfer to Handler

Figure 4-5. Stack Usage on Transfers to Interrupt and Exception Handling Routines

If a stack switch does occur, the processor does the following:

1. Temporarily saves (internally) the current contents of the SS, ESP, EFLAGS, CS, and EIP registers.

2. Loads the segment selector and stack pointer for the new stack (that is, the stack for the privilege level being called) from the TSS into the SS and ESP registers and switches to the new stack.
3. Pushes the temporarily saved SS, ESP, EFLAGS, CS, and EIP values for the interrupted procedure's stack onto the new stack.
4. Pushes an error code on the new stack (if appropriate).
5. Loads the segment selector for the new code segment and the new instruction pointer (from the interrupt gate or trap gate) into the CS and EIP registers, respectively.
6. If the call is through an interrupt gate, clears the IF flag in the EFLAGS register.
7. Begins execution of the handler procedure at the new privilege level.

A return from an interrupt or exception handler is initiated with the IRET instruction. The IRET instruction is similar to the far RET instruction, except that it also restores the contents of the EFLAGS register for the interrupted procedure:

When executing a return from an interrupt or exception handler from the same privilege level as the interrupted procedure, the processor performs these actions:

1. Restores the CS and EIP registers to their values prior to the interrupt or exception.
2. Restores the EFLAGS register.
3. Increments the stack pointer appropriately
4. Resumes execution of the interrupted procedure.

When executing a return from an interrupt or exception handler from a different privilege level than the interrupted procedure, the processor performs these actions:

1. Performs a privilege check.
2. Restores the CS and EIP registers to their values prior to the interrupt or exception.
3. Restores the EFLAGS register.
4. Restores the SS and ESP registers to their values prior to the interrupt or exception, resulting in a stack switch back to the stack of the interrupted procedure.
5. Resumes execution of the interrupted procedure.

4.4.2. Calls to Interrupt or Exception Handler Tasks

Interrupt and exception handler routines can also be executed in a separate task. Here, an interrupt or exception causes a task switch to a handler task. The handler task is given its own address space and (optionally) can execute at a higher protection level than application programs or tasks.

The switch to the handler task is accomplished with an implicit task call that references a **task gate descriptor**. The task gate provides access to the address space for the handler task. As part

of the task switch, the processor saves complete state information for the interrupted program or task. Upon returning from the handler task, the state of the interrupted program or task is restored and execution continues. See Chapter 5, *Interrupt and Exception Handling*, in the *Intel Architecture Software Developer's Manual, Volume 3*, for a detailed description of the processor's mechanism for handling interrupts and exceptions through handler tasks.

4.4.3. Interrupt and Exception Handling in Real-Address Mode

When operating in real-address mode, the processor responds to an interrupt or exception with an implicit far call to an interrupt or exception handler. The processor uses the interrupt or exception vector number as an index into an interrupt table. The interrupt table contains instruction pointers to the interrupt and exception handler procedures.

The processor saves the state of the EFLAGS register, the EIP register, the CS register, and an optional error code on the stack before switching to the handler procedure.

A return from the interrupt or exception handler is carried out with the IRET instruction.

See Chapter 15, *8086 Emulation*, in the *Intel Architecture Software Developer's Manual, Volume 3*, for more information on handling interrupts and exceptions in real-address mode.

4.4.4. INT *n*, INTO, INT 3, and BOUND Instructions

The INT *n*, INTO, INT 3, and BOUND instructions allow a program or task to explicitly call an interrupt or exception handler. The INT *n* instruction uses an interrupt vector as an argument, which allows a program to call any interrupt handler.

The INTO instruction explicitly calls the overflow exception (#OF) handler if the overflow flag (OF) in the EFLAGS register is set. The OF flag indicates overflow on arithmetic instructions, but it does not automatically raise an overflow exception. An overflow exception can only be raised explicitly in either of the following ways:

- Execute the INTO instruction.
- Test the OF flag and execute the INT *n* instruction with an argument of 4 (the vector number of the overflow exception) if the flag is set.

Both the methods of dealing with overflow conditions allow a program to test for overflow at specific places in the instruction stream.

The INT 3 instruction explicitly calls the breakpoint exception (#BP) handler.

The BOUND instruction explicitly calls the BOUND-range exceeded exception (#BR) handler if an operand is found to be not within predefined boundaries in memory. This instruction is provided for checking references to arrays and other data structures. Like the overflow exception, the BOUND-range exceeded exception can only be raised explicitly with the BOUND instruction or the INT *n* instruction with an argument of 5 (the vector number of the bounds-check exception). The processor does not implicitly perform bounds checks and raise the BOUND-range exceeded exception.

intel®

Intel Architecture Software Developer's Manual

Volume 2: Instruction Set Reference

NOTE: The *Intel Architecture Software Developer's Manual* consists of three volumes: *Basic Architecture*, Order Number 243190; *Instruction Set Reference*, Order Number 243191; and the *System Programming Guide*, Order Number 243192. Please refer to all three volumes when evaluating your design needs.

1997

G

Information in this document is provided in connection with Intel products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

Intel's Intel Architecture processors (e.g., Pentium® processor, Pentium processor with MMX™ technology, Pentium Pro processor, and Pentium II processor) may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an ordering number and are referenced in this document, or other Intel literature, may be obtained from:

Intel Corporation
P.O. Box 7641
Mt. Prospect IL 60056-7641

or call 1-800-879-4683
or visit Intel's website at http:\\www.intel.com

G

J*cc*—Jump if Condition Is Met (Continued)

Protected Mode Exceptions

#GP(0) If the offset being jumped to is beyond the limits of the CS segment.

Real-Address Mode Exceptions

#GP If the offset being jumped to is beyond the limits of the CS segment or is outside of the effective address space from 0 to FFFFH. This condition can occur if 32-address size override prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) If the offset being jumped to is beyond the limits of the CS segment or is outside of the effective address space from 0 to FFFFH. This condition can occur if 32-address size override prefix is used.

JMP—Jump

Opcode	Instruction	Description
EB *cb*	JMP *rel8*	Jump short, relative, displacement relative to next instruction
E9 *cw*	JMP *rel16*	Jump near, relative, displacement relative to next instruction
E9 *cd*	JMP *rel32*	Jump near, relative, displacement relative to next instruction
FF /4	JMP *r/m16*	Jump near, absolute indirect, address given in *r/m16*
FF /4	JMP *r/m32*	Jump near, absolute indirect, address given in *r/m32*
EA *cd*	JMP *ptr16:16*	Jump far, absolute, address given in operand
EA *cp*	JMP *ptr16:32*	Jump far, absolute, address given in operand
FF /5	JMP *m16:16*	Jump far, absolute indirect, address given in *m16:16*
FF /5	JMP *m16:32*	Jump far, absolute indirect, address given in *m16:32*

Description

Transfers program control to a different point in the instruction stream without recording return information. The destination (target) operand specifies the address of the instruction being jumped to. This operand can be an immediate value, a general-purpose register, or a memory location.

This instruction can be used to execute four different types of jumps:

- Near jump—A jump to an instruction within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment jump.
- Short jump—A near jump where the jump range is limited to –128 to +127 from the current EIP value.
- Far jump—A jump to an instruction located in a different segment than the current code segment but at the same privilege level, sometimes referred to as an intersegment jump.
- Task switch—A jump to an instruction located in a different task.

A task switch can only be executed in protected mode (see Chapter 6, *Task Management*, in the *Intel Architecture Software Developer's Manual, Volume 3*, for information on performing task switches with the JMP instruction).

Near and Short Jumps. When executing a near jump, the processor jumps to the address (within the current code segment) that is specified with the target operand. The target operand specifies either an absolute offset (that is an offset from the base of the code segment) or a relative offset (a signed displacement relative to the current value of the instruction pointer in the EIP register). A near jump to a relative offset of 8-bits (*rel8*) is referred to as a short jump. The CS register is not changed on near and short jumps.

An absolute offset is specified indirectly in a general-purpose register or a memory location (*r/m16* or *r/m32*). The operand-size attribute determines the size of the target operand (16 or 32 bits). Absolute offsets are loaded directly into the EIP register. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared to 0s, resulting in a maximum instruction pointer size of 16 bits.

JMP—Jump (Continued)

A relative offset (*rel8, rel16*, or *rel32*) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed 8-, 16-, or 32-bit immediate value. This value is added to the value in the EIP register. (Here, the EIP register contains the address of the instruction following the JMP instruction). When using relative offsets, the opcode (for short vs. near jumps) and the operand-size attribute (for near relative jumps) determines the size of the target operand (8, 16, or 32 bits).

Far Jumps in Real-Address or Virtual-8086 Mode. When executing a far jump in real-address or virtual-8086 mode, the processor jumps to the code segment and offset specified with the target operand. Here the target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). With the pointer method, the segment and address of the called procedure is encoded in the instruction, using a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. With the indirect method, the target operand specifies a memory location that contains a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address. The far address is loaded directly into the CS and EIP registers. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared to 0s.

Far Jumps in Protected Mode. When the processor is operating in protected mode, the JMP instruction can be used to perform the following three types of far jumps:

- A far jump to a conforming or non-conforming code segment.
- A far jump through a call gate.
- A task switch.

(The JMP instruction cannot be used to perform interprivilege level far jumps.)

In protected mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate, task gate, or TSS) and access rights determine the type of jump to be performed.

If the selected descriptor is for a code segment, a far jump to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far jump to the same privilege level in protected mode is very similar to one carried out in real-address or virtual-8086 mode. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register, and the offset from the instruction is loaded into the EIP register. Note that a call gate (described in the next paragraph) can also be used to perform far call to a code segment at the same privilege level. Using this mechanism provides an extra level of indirection and is the preferred method of making jumps between 16-bit and 32-bit code segments.

JMP—Jump (Continued)

When executing a far jump through a call gate, the segment selector specified by the target operand identifies the call gate. (The offset part of the target operand is ignored.) The processor then jumps to the code segment specified in the call gate descriptor and begins executing the instruction at the offset specified in the call gate. No stack switch occurs. Here again, the target operand can specify the far address of the call gate either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*).

Executing a task switch with the JMP instruction, is somewhat similar to executing a jump through a call gate. Here the target operand specifies the segment selector of the task gate for the task being switched to (and the offset part of the target operand is ignored). The task gate in turn points to the TSS for the task, which contains the segment selectors for the task's code and stack segments. The TSS also contains the EIP value for the next instruction that was to be executed before the task was suspended. This instruction pointer value is loaded into EIP register so that the task begins executing again at this next instruction.

The JMP instruction can also specify the segment selector of the TSS directly, which eliminates the indirection of the task gate. See Chapter 6, *Task Management*, in *Intel Architecture Software Developer's Manual, Volume 3*, the for detailed information on the mechanics of a task switch.

Note that when you execute at task switch with a JMP instruction, the nested task flag (NT) is not set in the EFLAGS register and the new TSS's previous task link field is not loaded with the old task's TSS selector. A return to the previous task can thus not be carried out by executing the IRET instruction. Switching tasks with the JMP instruction differs in this regard from the CALL instruction which does set the NT flag and save the previous task link information, allowing a return to the calling task with an IRET instruction.

Operation

```
IF near jump
   THEN IF near relative jump
      THEN
         tempEIP ← EIP + DEST; (* EIP is instruction following JMP instruction*)
      ELSE (* near absolute jump *)
         tempEIP ← DEST;
   FI;
   IF tempEIP is beyond code segment limit THEN #GP(0); FI;
   IF OperandSize = 32
      THEN
         EIP ← tempEIP;
      ELSE (* OperandSize=16 *)
         EIP ← tempEIP AND 0000FFFFH;
   FI;
FI:

IF far jump AND (PE = 0 OR (PE = 1 AND VM = 1)) (* real-address or virtual-8086 mode *)
   THEN
      tempEIP ← DEST(offset); (* DEST is ptr16:32 or [m16:32] *)
```

G

JMP—Jump (Continued)

```
        IF tempEIP is beyond code segment limit THEN #GP(0); FI;
        CS ← DEST(segment selector); (* DEST is ptr16:32 or [m16:32] *)
        IF OperandSize = 32
            THEN
                EIP ← tempEIP; (* DEST is ptr16:32 or [m16:32] *)
            ELSE (* OperandSize = 16 *)
                EIP ← tempEIP AND 0000FFFFH; (* clear upper 16 bits *)
        FI;
FI;
IF far jump AND (PE = 1 AND VM = 0) (* Protected mode, not virtual-8086 mode *)
    THEN
        IF effective address in the CS, DS, ES, FS, GS, or SS segment is illegal
            OR segment selector in target operand null
            THEN #GP(0);
        FI;
        IF segment selector index not within descriptor table limits
            THEN #GP(new selector);
        FI;
        Read type and access rights of segment descriptor;
        IF segment type is not a conforming or nonconforming code segment, call gate,
            task gate, or TSS THEN #GP(segment selector); FI;
        Depending on type and access rights
            GO TO CONFORMING-CODE-SEGMENT;
            GO TO NONCONFORMING-CODE-SEGMENT;
            GO TO CALL-GATE;
            GO TO TASK-GATE;
            GO TO TASK-STATE-SEGMENT;
    ELSE
        #GP(segment selector);
FI;

CONFORMING-CODE-SEGMENT:
    IF DPL > CPL THEN #GP(segment selector); FI;
    IF segment not present THEN #NP(segment selector); FI;
    tempEIP ← DEST(offset);
    IF OperandSize=16
        THEN tempEIP ← tempEIP AND 0000FFFFH;
    FI;
    IF tempEIP not in code segment limit THEN #GP(0); FI;
    CS ← DEST(SegmentSelector); (* segment descriptor information also loaded *)
    CS(RPL) ← CPL
    EIP ← tempEIP;
END;

NONCONFORMING-CODE-SEGMENT:
    IF (RPL > CPL) OR (DPL ≠ CPL) THEN #GP(code segment selector); FI;
```

JMP—Jump (Continued)

```
    IF segment not present THEN #NP(segment selector); FI;
    IF instruction pointer outside code segment limit THEN #GP(0); FI;
    tempEIP ← DEST(offset);
    IF OperandSize=16
        THEN tempEIP ← tempEIP AND 0000FFFFH;
    FI;
    IF tempEIP not in code segment limit THEN #GP(0); FI;
    CS ← DEST(SegmentSelector); (* segment descriptor information also loaded *)
    CS(RPL) ← CPL
    EIP ← tempEIP;
END;

CALL-GATE:
    IF call gate DPL < CPL
        OR call gate DPL < call gate segment-selector RPL
            THEN #GP(call gate selector); FI;
    IF call gate not present THEN #NP(call gate selector); FI;
    IF call gate code-segment selector is null THEN #GP(0); FI;
    IF call gate code-segment selector index is outside descriptor table limits
        THEN #GP(code segment selector); FI;
    Read code segment descriptor;
    IF code-segment segment descriptor does not indicate a code segment
        OR code-segment segment descriptor is conforming and DPL > CPL
        OR code-segment segment descriptor is non-conforming and DPL ≠ CPL
            THEN #GP(code segment selector); FI;
    IF code segment is not present THEN #NP(code-segment selector); FI;
    IF instruction pointer is not within code-segment limit THEN #GP(0); FI;
    tempEIP ← DEST(offset);
    IF GateSize=16
        THEN tempEIP ← tempEIP AND 0000FFFFH;
    FI;
    IF tempEIP not in code segment limit THEN #GP(0); FI;
    CS ← DEST(SegmentSelector); (* segment descriptor information also loaded *)
    CS(RPL) ← CPL
    EIP ← tempEIP;
END;

TASK-GATE:
    IF task gate DPL < CPL
        OR task gate DPL < task gate segment-selector RPL
            THEN #GP(task gate selector); FI;
    IF task gate not present THEN #NP(gate selector); FI;
    Read the TSS segment selector in the task-gate descriptor;
    IF TSS segment selector local/global bit is set to local
        OR index not within GDT limits
        OR TSS descriptor specifies that the TSS is busy
```

G

JMP—Jump (Continued)

```
            THEN #GP(TSS selector); FI;
   IF TSS not present THEN #NP(TSS selector); FI;
   SWITCH-TASKS to TSS;
   IF EIP not within code segment limit THEN #GP(0); FI;
END;

TASK-STATE-SEGMENT:
   IF TSS DPL < CPL
        OR TSS DPL < TSS segment-selector RPL
        OR TSS descriptor indicates TSS not available
            THEN #GP(TSS selector); FI;
   IF TSS is not present THEN #NP(TSS selector); FI;
   SWITCH-TASKS to TSS
   IF EIP not within code segment limit THEN #GP(0); FI;
END;
```

Flags Affected

All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur.

Protected Mode Exceptions

#GP(0) If offset in target operand, call gate, or TSS is beyond the code segment limits.

If the segment selector in the destination operand, call gate, task gate, or TSS is null.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.

#GP(selector) If segment selector index is outside descriptor table limits.

If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, call gate, task gate, or task state segment.

If the DPL for a nonconforming-code segment is not equal to the CPL

(When not using a call gate.) If the RPL for the segment's segment selector is greater than the CPL.

If the DPL for a conforming-code segment is greater than the CPL.

If the DPL from a call-gate, task-gate, or TSS segment descriptor is less than the CPL or than the RPL of the call-gate, task-gate, or TSS's segment selector.

JMP—Jump (Continued)

If the segment descriptor for selector in a call gate does not indicate it is a code segment.

If the segment descriptor for the segment selector in a task gate does not indicate available TSS.

If the segment selector for a TSS has its local/global bit set for local.

If a TSS segment descriptor specifies that the TSS is busy or not available.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NP (selector) If the code segment being accessed is not present.

If call gate, task gate, or TSS not present.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. (Only occurs when fetching target from memory.)

Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#GP(0) If the target operand is beyond the code segment limits.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made. (Only occurs when fetching target from memory.)

G

APPENDIX A OPCODE MAP

The opcode tables in this chapter are provided to aid in interpreting Intel Architecture object code. The instructions are divided into three encoding groups: 1-byte opcode encodings, 2-byte opcode encodings, and escape (floating-point) encodings. The 1- and 2-byte opcode encodings are used to encode integer, system, and MMX instructions. The opcode maps for these instructions are given in Tables A-1 through A-3. Sections A.2. through A.4. give instructions for interpreting 1- and 2-byte opcode maps. The escape encodings are used to encode floating-point instructions. The opcode maps for these instructions are given in Tables A-4 through A-13. Section A.5. gives instructions for interpreting the escape opcode maps.

See Chapter 2, *Instruction Format*, for detailed information on the ModR/M byte, register values, and the various addressing forms.

A.1. KEY TO ABBREVIATIONS

Operands are identified by a two-character code of the form Zz. The first character, an uppercase letter, specifies the addressing method; the second character, a lowercase letter, specifies the type of operand.

A.1.1. Codes for Addressing Method

The following abbreviations are used for addressing methods:

A Direct address. The instruction has no ModR/M byte; the address of the operand is encoded in the instruction; and no base register, index register, or scaling factor can be applied, for example, far JMP (EA).

C The reg field of the ModR/M byte selects a control register, for example, MOV (0F20, 0F22).

D The reg field of the ModR/M byte selects a debug register, for example, MOV (0F21,0F23).

E A ModR/M byte follows the opcode and specifies the operand. The operand is either a general-purpose register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, a displacement.

F EFLAGS Register.

G The reg field of the ModR/M byte selects a general register, for example, AX (000).

I Immediate data. The operand value is encoded in subsequent bytes of the instruction.

J The instruction contains a relative offset to be added to the instruction pointer register, for example, JMP short, LOOP.

M The ModR/M byte may refer only to memory, for example, BOUND, LES, LDS, LSS, LFS, LGS, CMPXCHG8B.

O The instruction has no ModR/M byte; the offset of the operand is coded as a word or double word (depending on address size attribute) in the instruction. No base register, index register, or scaling factor can be applied, for example, MOV (A0–A3).

P The reg field of the ModR/M byte selects a packed quadword MMX register.

Q An ModR/M byte follows the opcode and specifies the operand. The operand is either an MMX register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, and a displacement.

R The mod field of the ModR/M byte may refer only to a general register, for example, MOV (0F20-0F24, 0F26).

S The reg field of the ModR/M byte selects a segment register, for example, MOV (8C,8E).

T The reg field of the ModR/M byte selects a test register, for example, MOV (0F24,0F26).

X Memory addressed by the DS:SI register pair (for example, MOVS, OUTS, or LODS).

Y Memory addressed by the ES:DI register pair (for example, MOVS, INS, or STOS).

A.1.2. Codes for Operand Type

The following abbreviations are used for operand types:

a Two one-word operands in memory or two double-word operands in memory, depending on operand size attribute (used only by the BOUND instruction).

b Byte, regardless of operand-size attribute.

c Byte or word, depending on operand-size attribute.

d Doubleword, regardless of operand-size attribute.

p 32-bit or 48-bit pointer, depending on operand size attribute.

q Quadword, regardless of operand-size attribute.

s 6-byte pseudo-descriptor.

v Word or doubleword, depending on operand-size attribute.

w Word, regardless of operand-size attribute.

G

A.1.3. Register Codes

When an operand is a specific register encoded in the opcode, the register is identified by its name (for example, AX, CL, or ESI). The name of the register indicates whether the register is 32, 16, or 8 bits wide. A register identifier of the form eXX is used when the width of the register depends on the operand size attribute. For example, eAX indicates that the AX register is used when the operand size attribute is 16, and the EAX register is used when the operand size attribute is 32.

A.2. ONE-BYTE OPCODE INTEGER INSTRUCTIONS

The opcode map for 1-byte opcodes are shown in Table A-1. For 1-byte opcodes, the instruction and its operands can be determined from the hexadecimal opcode. For example, the opcode 030500000000H for an ADD instruction can be interpreted from the 1-byte opcode map in Table A-1 as follows. The first digit (0) of the opcode indicates the row and the second digit (3) indicates the column in the opcode map.table, which points to ADD instruction with operand types Gv and Ev. The first operand (type Gv) indicates a general register that is a word or doubleword depending on the operand-size attribute. The second operand (type Ev) indicates that a ModR/M byte follows that specifies whether the operand is a word or doubleword general-purpose register or a memory address. The ModR/M byte for this instruction is 05H, which indicates that a 32-bit displacement follows (00000000H). The reg/opcode portion of the ModR/M byte (bits 3 through 5) is 000 indicating the EAX register. Thus, it can be determined that the instruction for this opcode is ADD EAX, mem_op and the offset of mem_op is 00000000H.

Some 1- and 2-byte opcodes point to "group" numbers. These group numbers indicate that the instruction uses the reg/opcode bits in the ModR/M byte as an opcode extension (see Section A.4., "Opcode Extensions For One- And Two-byte Opcodes").

A.3. TWO-BYTE OPCODE INTEGER INSTRUCTIONS

Instructions that begin with 0FH can be found in the two-byte opcode map given in Table A-2. Here, the second opcode byte is used to reference a row and column in the Table. For example, the opcode 0FA4050000000003H is located on the first page of the two-byte opcode map in row A, column 4, which points to an SHLD instruction with the operands Ev, Gv, and Ib. The Ev, Gv, and Ib operands are interpreted as follows. The first operand (Ev type) indicates that a ModR/M byte follows the opcode to specify a word or doubleword operand. The second operand (Gv type) indicates that the reg field of the ModR/M byte selects a general-purpose register. The third operand (Ib type) indicates that immediate data is encoded in the subsequent byte of the instruction.

The third byte of the opcode (05H) is the ModR/M byte. The mod and opcode/reg fields indicate that a 32-bit displacement follows and that the EAX register is the source.

Table A-1. One-Byte Opcode Map[1]

	0	1	2	3	4	5	6	7
0	ADD						PUSH	POP
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	ES	ES
1	ADC						PUSH	POP
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	SS	SS
2	AND						SEG	DAA
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	=ES	
3	XOR						SEG	AAA
	Eb,Gb	Ev,Gv	Gb,Eb	Gb,Ev	AL,Ib	eAX,Iv	=SS	
4	INC general register							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
5	PUSH general register							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
6	PUSHA	POPA	BOUND	ARPL	SEG	SEG	Operand	Address
	PUSHAD	POPAD	Gv,Ma	Ew,Gw	=FS	=GS	Size	Size
7.	Short-displacement jump on condition (Jb)							
	JO	JNO	JB/JNAE/JC	JNB/JAE/JNC	JZ	JNZ	JBE	JNBE
8	Immediate Group 1^2				TEST		XCHG	
	Eb,Ib	Ev,Iv	Ev,Ib	Eb,Ib	Eb,Gb	Ev,Gv	Eb,Gb	Ev,Gv
9	NOP	XCHG word or double-word register with eAX						
		eCX	eDX	eBX	eSP	eBP	eSI	eDI
A	MOV				MOVSB	MOVSW	CMPSB	CMPSW
	AL,Ob	eAX,Ov	Ob,AL	Ov,eAX	Xb,Yb	Xv,Yv	Xb,Yb	Xv,Yv
B	MOV immediate byte into byte register							
	AL	CL	DL	BL	AH	CH	DH	BH
C	Shift Group 2a^2		RET near		LES	LDS	MOV	
	Eb,Ib	Ev,Ib	Iw		Gv,Mp	Gv,Mp	Eb,Ib	Ev,Iv
D	Shift Group 2^2				AAM	AAD		XLAT
	Eb,1	Ev,1	Eb,CL	Ev,CL				
E	LOOPN	LOOPE	LOOP	JCXZ/JECXZ	IN		OUT	
	Jb	Jb	Jb	Jb	AL,Ib	eAX,Ib	Ib,AL	Ib,eAX
F	LOCK		REPNE	REP	HLT	CMC	Unary Group 3^2	
				REPE			Eb	Ev

Table A-1. One-Byte Opcode Map (Continued)

	8	9	A	B	C	D	E	F
0	OR						PUSH	2-byte
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	CS	Escape
1	SBB						PUSH	POP
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	DS	DS
2	SUB						SEG	DAS
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	=CS	
3	CMP	SEG	AAS					
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv	=DS	
4	DEC General-Purpose Register							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
5	POP Into General-Purpose Register							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
6	PUSH	IMUL	PUSH	IMUL	INSB	INSW/D	OUTSB	OUTSW/D
	Iv	Gv,Ev,Iv	Ib	Gv,Ev,Ib	Yb,DX	Yv,DX	Dx,Xb	DX,Xv
7	Short-Displacement Jump on Condition (Jb)							
	JS	JNS	JP	JNP	JL	JNL	JLE	JNLE
8	MOV				MOV	LEA	MOV	POP
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	Ew,Sw	Gv,M	Sw,Ew	Ev
9	CBW	CWD/CDQ	CALL	WAIT	PUSHF	POP	SAHF	LAHF
			aP		Fv	Fv		
A	TEST		STOSB	STOSW/D	LODSB	LODSW/D	SCASB	SCASW/D
	AL,Ib	eAX,Iv	Yb,AL	Yv,eAX	AL,Xb	eAX,Xv	AL,Yb	eAX,Yv
B	MOV Immediate Word or Double Into Word or Double Register							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
C	ENTER	LEAVE	RET far	RET far	INT	INT	INTO	IRET
	Iw, Ib		Iw		3	Ib		
D	ESC (Escape to Coprocessor Instruction Set)							
E	CALL	JMP			IN		OUT	
	Jv	Jv	Ap	Jb	AL,DX	eAX,DX	DX,AL	DX,eAX
F	CLC	STC	CLI	STI	CLD	STD	INC/DEC	INC/DEC
							Group 4^2	Group 5^2

NOTES:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.
2. Bits 5, 4, and 3 of ModR/M byte used as an opcode extension (see Section A.4.).

The next part of the SHLD opcode is the 32-bit displacement for the destination memory operand (00000000H), which is followed by the immediate byte representing the count of the shift (03H). By this breakdown, it has been shown that the opcode 0FA4050000000003H represents the instruction: SHLD DS:00000000H, EAX, 3.

Table A-2. Two Byte Opcode Map (First byte is 0FH)[1]

	0	1	2	3	4	5	6	7
0	Group 6[2]		LAR Gv,Ew	LSL Gv,Ew			CLTS	
1								
2	MOV Rd,Cd	MOV Rd,Dd	MOV Cd,Rd	MOV Dd,Rd				
3	WRMSR	RDTSC	RDMSR	RDPMC				
4	CMOVO Gv, Ev	CMOVNO Gv, Ev	CMOVB, CMOVC, CMOVNAE Gv, Ev	CMOVAE, CMOVNB, CMOVNC Gv, Ev	CMOVE, CMOVZ Gv, Ev	CMOVNE, CMOVNZ Gv, Ev	CMOVBE, CMOVNA Gv, Ev	CMOVA, CMOVNBE Gv, Ev
5								
6	PUNPCKLBW Pq, Qd	PUNPCKLWD Pq, Qd	PUNOCKLDQ Pq, Qd	PACKUSDW Pq, Qd	PCMPGTB Pq, Qd	PCMPGTW Pq, Qd	PCMPGTD Pq, Qd	PACKSSWB Pq, Qd
		Group A[2]			PCMPEQB Pq, Qd	PCMPEQW Pq, Qd	PCMPEQD Pq, Qd	EMMS
7		PSHIMW[3]	PSHIMD[3]	PSHIMQ[3]				
8	Long-Displacement Jump on Condition (Jv)							
	JO	JNO	JB	JNB	JZ	JNZ	JBE	JNBE
9	Byte Set on condition (Eb)							
	SETO	SETNO	SETB	SETNB	SETZ	SETNZ	SETBE	SETNBE
A	PUSH FS	POP FS	CPUID	BT Ev,Gv	SHLD Ev,Gv,Ib	SHLD Ev,Gv,CL		
B	CMPXCHG	CMPXCHG	LSS	BTR	LFS	LGS	MOVZX	
	Eb,Gb	Ev,Gv	Mp	Ev,Gv	Mp	Mp	Gv,Eb	Gv,Ew
C	XADD Eb,Gb	XADD Ev,Gv						Group 9[2]
D		PSRLW Pq, Qd	PSRLD Pq, Qd	PSRLQ Pq, Qd		PMULLW Pq, Qd		
E		PSRAW Pq, Qd	PSRAD Pq, Qd	- ---		PMULHW Pq, Qd		
F		PSLLW Pq, Qd	PSLLD Pq, Qd	PSLLQ Pq, Qd		PMADDWD Pq, Qd		

Table A-2. Two-Byte Opcode Map (First byte is 0FH) (Continued)

	8	9	A	B	C	D		
0	INVD	WBINVD		UD2[4]				
1								
2								
3								
4	CMOVS Gv, Ev	CMOVNS Gv, Ev	CMOVP, CMOVPE Gv, Ev	CMOVNP, CMOVPO Gv, Ev	CMOVL, CMOVNGE Gv, Ev	CMOVGE, CMOVNL Gv, Ev	CMOVLE, CMOVNG Gv, Ev	CMOVG, CMOVNLE Gv, Ev
5								
6	PUNPCKHBW Pq, Qd	PUNPCKHWD Pq, Qd	PUNPCKHDQ Pq, Qd	PACKSSDW Pq, Qd			MOVD Pd, Ed	MOVQ Pq, Qq
7							MOVD Ed, Pd	MOVQ Qq, Pq
8	Long-Displacement Jump on Condition (Jv)							
	JS	JNS	JP	JNP	JL	JNL	JLE	JNLE
	Byte set on condition (Eb)							
9	SETS	SETNS	SETP	SETNP	SETL	SETNL	SETLE	SETNLE
	Eb	Eb	Eb	Eb	Eb	Eb	Eb	Eb
A	PUSH GS	POP GS	RSM	BTS Ev,Gv	SHRD Ev,Gv,Ib	SHRD Ev,Gv,CL		IMUL Gv,Ev
B		Invalid Opcode[4]	Group 8[2]	BTC	BSF	BSR	MOVSX	
			Ev,Ib	Ev,Gv	Gv,Ev	Gv,Ev	Gv,Eb	Gv,Ew
C	BSWAP EAX	BSWAP ECX	BSWAP EDX	BSWAP EBX	BSWAP ESP	BSWAP EBP	BSWAP ESI	BSWAP EDI
D	PSUBUSB Pq, Qq	PSUBUSW Pq, Qq		PAND Pq, Qq	PADDUSB Pq, Qq	PADDUSW Pq, Qq		PANDN Pq, Qq
E	PSUBSB Pq, Qq	PSUBSW Pq, Qq		POR Pq, Qq	PADDSB Pq, Qq	PADDSW Pq, Qq		PXOR Pq, Qq
F	PSUBB Pq, Qq	PSUBW Pq, Qq	PSUBD Pq, Qq		PADDB Pq, Qq	PADDW Pq, Qq	PADDD Pq, Qq	

NOTES:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.
2. Bits 5, 4, and 3 of ModR/M byte used as an opcode extension (see Section A.4.).
3. These abbreviations are not actual mnemonics. When shifting by immediate shift counts, the PSHIMD mnemonic represents the PSLLD, PSRAD, and PSRLD instructions, PSHIMW represents the PSLLW, PSRAW, and PSRLW instructions, and PSHIMQ represents the PSLLQ and PSRLQ instructions. The instructions that shift by immediate counts are differentiated by the ModR/M bytes (see Section A.4.).
4. Use the 0F0B opcode (UD2 instruction) or the 0FB9H opcode when deliberately trying to generate an invalid opcode exception (#UD).

9

intel®

Intel Architecture Software Developer's Manual

Volume 3: System Programming Guide

NOTE: The *Intel Architecture Developer's Manual* consists of three books: *Basic Architecture*, Order Number 243190; *Instruction Set Reference Manual*, Order Number 243191; and the *System Programming Guide*, Order Number 243192.
Please refer to all three volumes when evaluating your design needs.

1997

G

Information in this document is provided in connection with Intel products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

Intel's Intel Architecture processors (e.g., Pentium® processor, Pentium Pro processor, and Pentium II processor) may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an ordering number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725 or by visiting Intel's website at http://www.intel.com

G

CHAPTER 3
PROTECTED-MODE MEMORY MANAGEMENT

This chapter describes the Intel Architecture's protected-mode memory management facilities, including the physical memory requirements, the segmentation mechanism, and the paging mechanism. See Chapter 4, *Protection*, for a description of the processor's protection mechanism. See Chapter 15, *8086 Emulation*, for a description of memory addressing protection in real-address and virtual-8086 modes.

3.1. MEMORY MANAGEMENT OVERVIEW

The memory management facilities of the Intel Architecture are divided into two parts: segmentation and paging. Segmentation provides a mechanism of isolating individual code, data, and stack modules so that multiple programs (or tasks) can run on the same processor without interfering with one another. Paging provides a mechanism for implementing a conventional demand-paged, virtual-memory system where sections of a program's execution environment are mapped into physical memory as needed. Paging can also be used to provide isolation between multiple tasks. When operating in protected mode, some form of segmentation must be used. **There is no mode bit to disable segmentation.** The use of paging, however, is optional.

These two mechanisms (segmentation and paging) can be configured to support simple single-program (or single-task) systems, multitasking systems, or multiple-processor systems that used shared memory.

As shown in Figure 3-1, segmentation provides a mechanism for dividing the processor's addressable memory space (called the **linear address space**) into smaller protected address spaces called **segments**. Segments can be used to hold the code, data, and stack for a program or to hold system data structures (such as a TSS or LDT). If more than one program (or task) is running on a processor, each program can be assigned its own set of segments. The processor then enforces the boundaries between these segments and insures that one program does not interfere with the execution of another program by writing into the other program's segments. The segmentation mechanism also allows typing of segments so that the operations that may be performed on a particular type of segment can be restricted.

All of the segments within a system are contained in the processor's linear address space. To locate a byte in a particular segment, a **logical address** (sometimes called a far pointer) must be provided. A logical address consists of a segment selector and an offset. The segment selector is a unique identifier for a segment. Among other things it provides an offset into a descriptor table (such as the global descriptor table, GDT) to a data structure called a segment descriptor. Each segment has a segment descriptor, which specifies the size of the segment, the access rights and privilege level for the segment, the segment type, and the location of the first byte of the segment in the linear address space (called the base address of the segment). The offset part of the logical address is added to the base address for the segment to locate a byte within the segment. The base address plus the offset thus forms a **linear address** in the processor's linear address space.

9

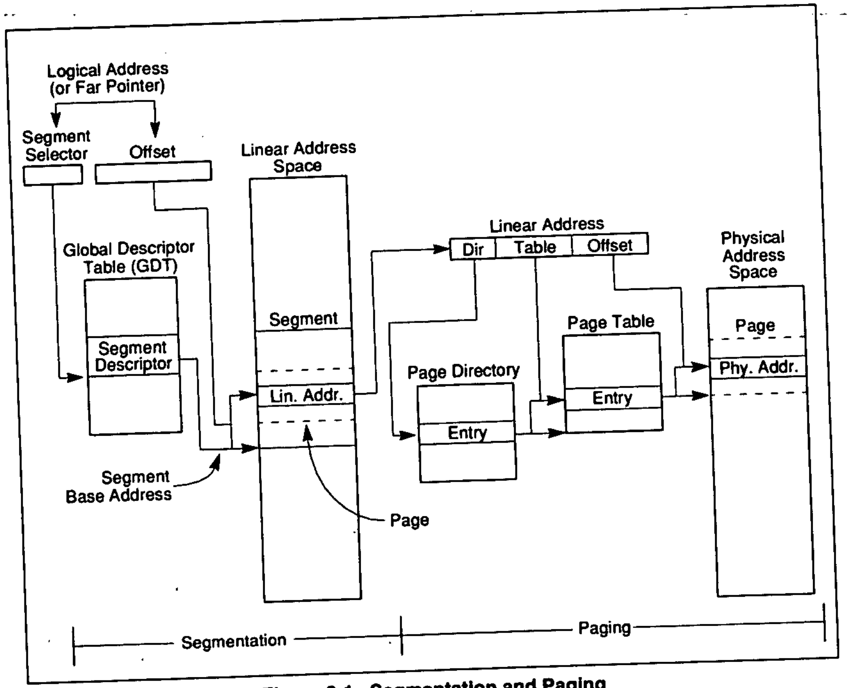


Figure 3-1. Segmentation and Paging

If paging is not used, the linear address space of the processor is mapped directly into the physical address space of processor. The physical address space is defined as the range of addresses that the processor can generate on its address bus.

Because multitasking computing systems commonly define a linear address space much larger than it is economically feasible to contain all at once in physical memory, some method of "virtualizing" the linear address space is needed. This virtualization of the linear address space is handled through the processor's paging mechanism.

Paging supports a "virtual memory" environment where a large linear address space is simulated with a small amount of physical memory (RAM and ROM) and some disk storage. When using paging, each segment is divided into pages (ordinarily 4 KBytes each in size), which are stored either in physical memory or on the disk. The operating system or executive maintains a page directory and a set of page tables to keep track of the pages. When a program (or task) attempts to access an address location in the linear address space, the processor uses the page directory and page tables to translate the linear address into a physical address and then performs the requested operation (read or write) on the memory location. If the page being accessed is not currently in physical memory, the processor interrupts execution of the program (by generating a page-fault exception). The operating system or executive then reads the page into physical memory from the disk and continues executing the program.

When paging is implemented properly in the operating-system or executive, the swapping of pages between physical memory and the disk is transparent to the correct execution of a program. Even programs written for 16-bit Intel Architecture processors can be paged (transparently) when they are run in virtual-8086 mode.

3.2. USING SEGMENTS

The segmentation mechanism supported by the Intel Architecture can be used to implement a wide variety of system designs. These designs range from flat models that make only minimal use of segmentation to protect programs to multisegmented models that employ segmentation to create a robust operating environment in which multiple programs and tasks can be executed reliably.

The following sections give several examples of how segmentation can be employed in a system to improve memory management performance and reliability.

3.2.1. Basic Flat Model

The simplest memory model for a system is the basic "flat model," in which the operating system and application programs have access to a continuous, unsegmented address space. To the greatest extent possible, this basic flat model hides the segmentation mechanism of the architecture from both the system designer and the application programmer.

To implement a basic flat memory model with the Intel Architecture, at least two segment descriptors must be created, one for referencing a code segment and one for referencing a data segment (see Figure 3-2). Both of these segments, however, are mapped to the entire linear address space: that is, both segment descriptors have the same base address value of 0 and the same segment limit of 4 GBytes. By setting the segment limit to 4 GBytes, the segmentation mechanism is kept from generating exceptions for out of limit memory references, even if no physical memory resides at a particular address. ROM (EPROM) is generally located at the top of the physical address space, because the processor begins execution at FFFF_FFF0H. RAM (DRAM) is placed at the bottom of the address space because the initial base address for the DS data segment after reset initialization is 0.

9

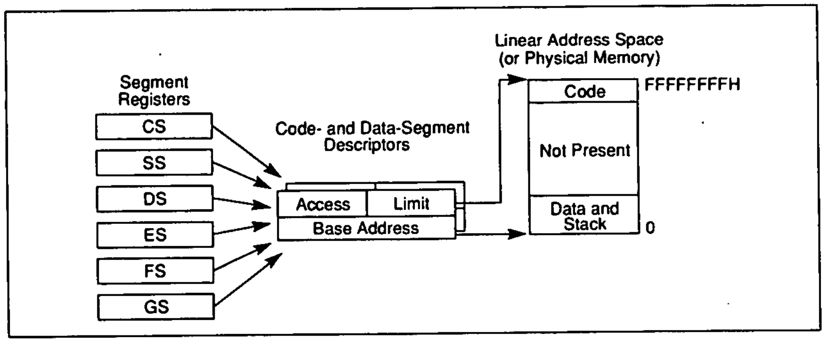


Figure 3-2. Flat Model

3.2.2. Protected Flat Model

The protected flat model is similar to the basic flat model, except the segment limits are set to include only the range of addresses for which physical memory actually exists (see Figure 3-3). A general-protection exception (#GP) is then generated on any attempt to access nonexistent memory. This model provides a minimum level of hardware protection against some kinds of program bugs.

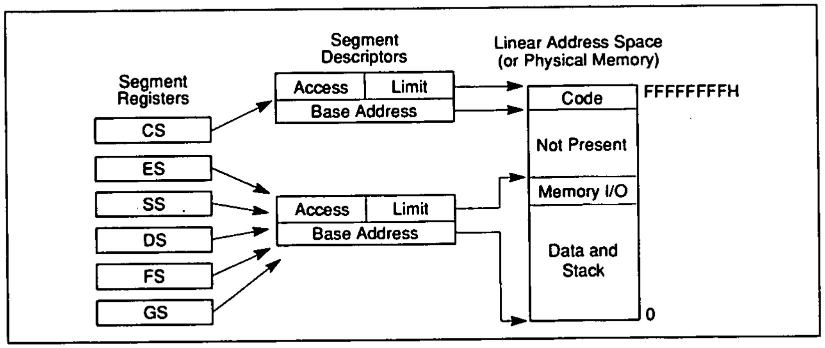


Figure 3-3. Protected Flat Model

More complexity can be added to this protected flat model to provide more protection. For example, for the paging mechanism to provide isolation between user and supervisor code and data, four segments need to be defined: code and data segments at privilege level 3 for the user, and code and data segments at privilege level 0 for the supervisor. Usually these segments all overlay each other and start at address 0 in the linear address space. This flat segmentation

model along with a simple paging structure can protect the operating system from applications, and by adding a separate paging structure for each task or process, it can also protect applications from each other. Similar designs are used by several popular multitasking operating systems.

3.2.3. Multisegment Model

A multisegment model (such as the one shown in Figure 3-4) uses the full capabilities of the segmentation mechanism to provided hardware enforced protection of code, data structures, and programs and tasks. Here, each program (or task) is given its own table of segment descriptors and its own segments. The segments can be completely private to their assigned programs or shared among programs. Access to all segments and to the execution environments of individual programs running on the system is controlled by hardware.

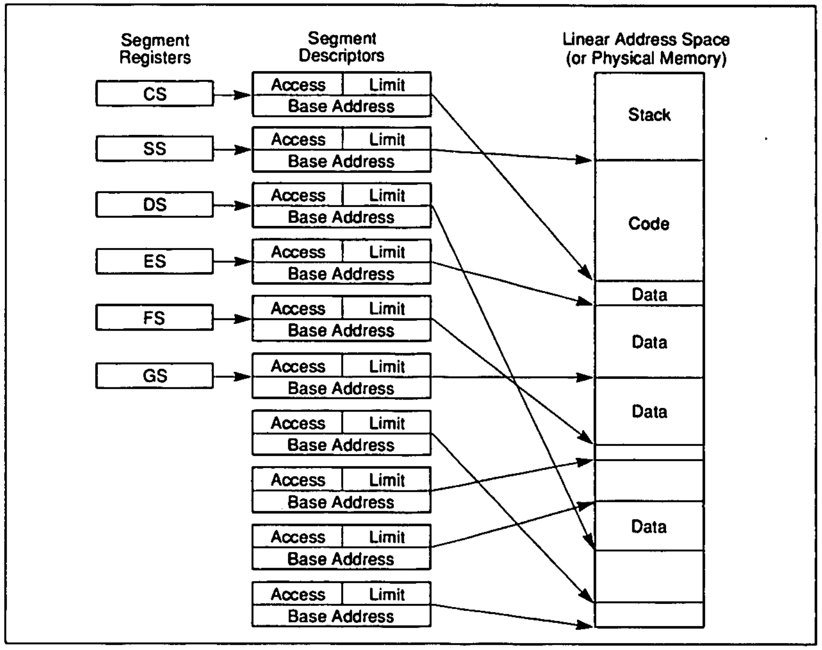


Figure 3-4. Multisegment Model

G

Access checks can be used to protect not only against referencing an address outside the limit of a segment, but also against performing disallowed operations in certain segments. For example, since code segments are designated as read-only segments, hardware can be used to prevent writes into code segments. The access rights information created for segments can also be used to set up protection rings or levels. Protection levels can be used to protect operating-system procedures from unauthorized access by application programs.

3.2.4. Paging and Segmentation

Paging can be used with any of the segmentation models described in Figures 3-2, 3-3, and 3-4. The processor's paging mechanism divides the linear address space (into which segments are mapped) into pages (as shown in Figure 3-1). These linear-address-space pages are then mapped to pages in the physical address space. The paging mechanism offers several page-level protection facilities that can be used with or instead of the segment-protection facilities. For example, it lets read-write protection be enforced on a page-by-page basis. The paging mechanism also provides two-level user-supervisor protection that can also be specified on a page-by-page basis.

3.3. PHYSICAL ADDRESS SPACE

In protected mode, the Intel Architecture provides a normal physical address space of 4 GBytes (2^{32}bytes). This is the address space that the processor can address on its address bus. This address space is flat (unsegmented), with addresses ranging continuously from 0 to FFFFFFFFH. This physical address space can be mapped to read-write memory, read-only memory, and memory mapped I/O. The memory mapping facilities described in this chapter can be used to divide this physical memory up into segments and/or pages.

(Introduced in the Pentium Pro processor.) The Intel Architecture also supports an extension of the physical address space to 2^{36} bytes (64 GBytes), with a maximum physical address of FFFFFFFFFH. This extension is invoked with the physical address extension (PAE) flag, located in bit 5 of control register CR4. (See Section 3.8., "Physical Address Extension", for more information about extended physical addressing.)

3.4. LOGICAL AND LINEAR ADDRESSES

At the system-architecture level in protected mode, the processor uses two stages of address translation to arrive at a physical address: logical-address translation and linear address space paging.

Even with the minimum use of segments, every byte in the processor's address space is accessed with a logical address. A logical address consists of a 16-bit segment selector and a 32-bit offset (see Figure 3-5). The segment selector identifies the segment the byte is located in and the offset specifies the location of the byte in the segment relative to the base address of the segment.

The processor translates every logical address into a linear address. A linear address is a 32-bit address in the processor's linear address space. Like the physical address space, the linear address space is a flat (unsegmented), 2^{32}-byte address space, with addresses ranging from 0 to FFFFFFFFH. The linear address space contains all the segments and system tables defined for a system.

To translate a logical address into a linear address, the processor does the following:

1. Uses the offset in the segment selector to locate the segment descriptor for the segment in the GDT or LDT and reads it into the processor. (This step is needed only when a new segment selector is loaded into a segment register.)
2. Examines the segment descriptor to check the access rights and range of the segment to insure that the segment is accessible and that the offset is within the limits of the segment.
3. Adds the base address of the segment from the segment descriptor to the offset to form a linear address.

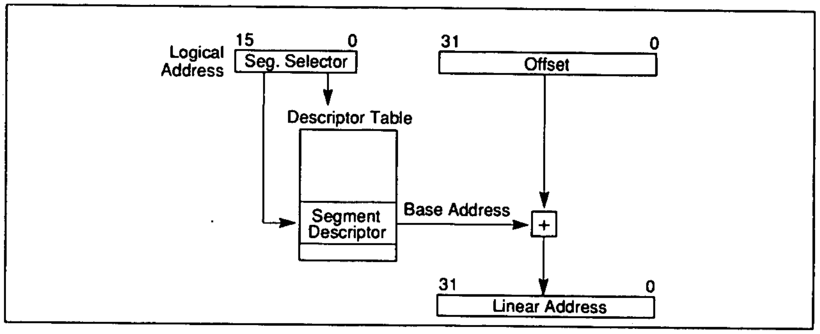


Figure 3-5. Logical Address to Linear Address Translation

If paging is not used, the processor maps the linear address directly to a physical address (that is, the linear address goes out on the processor's address bus). If the linear address space is paged, a second level of address translation is used to translate the linear address into a physical address. Page translation is described in Section 3.6., "Paging (Virtual Memory)".

3.4.1. Segment Selectors

A segment selector is a 16-bit identifier for a segment (see Figure 3-6). It does not point directly to the segment, but instead points to the segment descriptor that defines the segment. A segment selector contains the following items:

Index (Bits 3 through 15). Selects one of 8192 descriptors in the GDT or LDT. The processor multiplies the index value by 8 (the number of bytes in a segment descriptor) and adds the result to the base address of the GDT or LDT (from the GDTR or LDTR register, respectively).

TI (table indicator) flag
(Bit 2). Specifies the descriptor table to use: clearing this flag selects the GDT; setting this flag selects the current LDT.

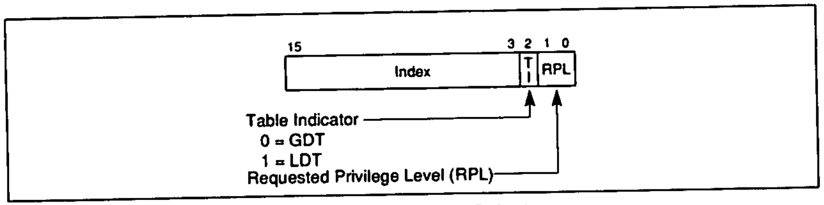


Figure 3-6. Segment Selector

Requested Privilege Level (RPL)
(Bits 0 and 1). Specifies the privilege level of the selector. The privilege level can range from 0 to 3, with 0 being the most privileged level. See Section 4.5., "Privilege Levels", for a description of the relationship of the RPL to the CPL of the executing program (or task) and the descriptor privilege level (DPL) of the descriptor the segment selector points to.

The first entry of the GDT is not used by the processor. A segment selector that points to this entry of the GDT (that is, a segment selector with an index of 0 and the TI flag set to 0) is used as a "null segment selector." The processor does not generate an exception when a segment register (other than the CS or SS registers) is loaded with a null selector. It does, however, generate an exception when a segment register holding a null selector is used to access memory. A null selector can be used to initialize unused segment registers. Loading the CS or SS register with a null segment selector causes a general-protection exception (#GP) to be generated.

Segment selectors are visible to application programs as part of a pointer variable, but the values of selectors are usually assigned or modified by link editors or linking loaders, not application programs.

3.4.2. Segment Registers

To reduce address translation time and coding complexity, the processor provides registers for holding up to 6 segment selectors (see Figure 3-7). Each of these segment registers support a specific kind of memory reference (code, stack, or data). For virtually any kind of program execution to take place, at least the code-segment (CS), data-segment (DS), and stack-segment (SS) registers must be loaded with valid segment selectors. The processor also provides three additional data-segment registers (ES, FS, and GS), which can be used to make additional data segments available to the currently executing program (or task).

For a program to access a segment, the segment selector for the segment must have been loaded in one of the segment registers. So, although a system can define thousands of segments, only 6 can be available for immediate use. Other segments can be made available by loading their segment selectors into these registers during program execution.

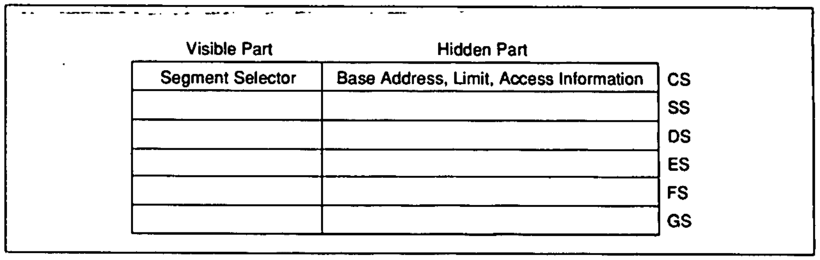


Figure 3-7. Segment Registers

Every segment register has a "visible" part and a "hidden" part. (The hidden part is sometimes referred to as a "descriptor cache" or a "shadow register.") When a segment selector is loaded into the visible part of a segment register, the processor also loads the hidden part of the segment register with the base address, segment limit, and access control information from the segment descriptor pointed to by the segment selector. The information cached in the segment register (visible and hidden) allows the processor to translate addresses without taking extra bus cycles to read the base address and limit from the segment descriptor. In systems in which multiple processors have access to the same descriptor tables, it is the responsibility of software to reload the segment registers when the descriptor tables are modified. If this is not done, an old segment descriptor cached in a segment register might be used after its memory-resident version has been modified.

Two kinds of load instructions are provided for loading the segment registers:

1. Direct load instructions such as the MOV, POP, LDS, LES, LSS, LGS, and LFS instructions. These instructions explicitly reference the segment registers.
2. Implied load instructions such as the far pointer versions of the CALL, JMP, and RET instructions and the IRET, INT*n*, INTO and INT3 instructions. These instructions change the contents of the CS register (and sometimes other segment registers) as an incidental part of their operation.

The MOV instruction can also be used to store visible part of a segment register in a general-purpose register.

3.4.3. Segment Descriptors

A segment descriptor is a data structure in a GDT or LDT that provides the processor with the size and location of a segment, as well as access control and status information. Segment descriptors are typically created by compilers, linkers, loaders, or the operating system or executive, but not application programs. Figure 3-8 illustrates the general descriptor format for all types of segment descriptors.

9

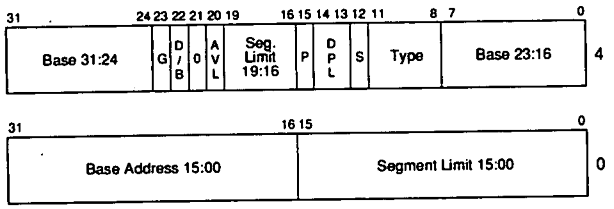
The flags and fields in a segment descriptor are as follows:

Segment limit field

Specifies the size of the segment. The processor puts together the two segment limit fields to form a 20-bit value. The processor interprets the segment limit in one of two ways, depending on the setting of the G (granularity) flag:

- If the granularity flag is clear, the segment size can range from 1 byte to 1 MByte, in byte increments.
- If the granularity flag is set, the segment size can range from 4 KBytes to 4 GBytes, in 4-KByte increments.

The processor uses the segment limit in two different ways, depending on whether the segment is an expand-up or an expand-down segment. See Section 3.4.3.1., "Code- and Data-Segment Descriptor Types", for more information about segment types. For expand-up segments, the offset in a logical address can range from 0 to the segment limit. Offsets greater than the segment limit generate general-protection exceptions (#GP). For expand-down segments, the segment limit has the reverse function; the offset can range from the segment limit to FFFFFFFFH or FFFFH, depending on the setting of the B flag. Offsets less than the segment limit generate general-protection exceptions. Decreasing the value in the segment limit field for an expand-down segment allocates new memory at the bottom of the segment's address space, rather than at the top. Intel Architecture stacks always grow downwards, making this mechanism is convenient for expandable stacks.



AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

Figure 3-8. Segment Descriptor

Base address fields

Defines the location of byte 0 of the segment within the 4-GByte linear address space. The processor puts together the three base address fields to form a single 32-bit value. Segment base addresses should be aligned to 16-byte boundaries. Although 16-byte alignment is not required, this alignment allows programs to maximize performance by aligning code and data on 16-byte boundaries.

Type field Indicates the segment or gate type and specifies the kinds of access that can be made to the segment and the direction of growth. The interpretation of this field depends on whether the descriptor type flag specifies an application (code or data) descriptor or a system descriptor. The encoding of the type field is different for code, data, and system descriptors (see Figure 4-1). See Section 3.4.3.1., "Code- and Data-Segment Descriptor Types", for a description of how this field is used to specify code and data-segment types.

S (descriptor type) flag

Specifies whether the segment descriptor is for a system segment (S flag is clear) or a code or data segment (S flag is set).

DPL (descriptor privilege level) field

Specifies the privilege level of the segment. The privilege level can range from 0 to 3, with 0 being the most privileged level. The DPL is used to control access to the segment. See Section 4.5., "Privilege Levels", for a description of the relationship of the DPL to the CPL of the executing code segment and the RPL of a segment selector.

P (segment-present) flag

Indicates whether the segment is present in memory (set) or not present (clear). If this flag is clear, the processor generates a segment-not-present exception (#NP) when a segment selector that points to the segment descriptor is loaded into a segment register. Memory management software can use this flag to control which segments are actually loaded into physical memory at a given time. It offers a control in addition to paging for managing virtual memory.

Figure 3-9 shows the format of a segment descriptor when the segment-present flag is clear. When this flag is clear, the operating system or executive is free to use the locations marked "Available" to store its own data, such as information regarding the whereabouts of the missing segment.

D/B (default operation size/default stack pointer size and/or upper bound) flag

Performs different functions depending on whether the segment descriptor is an executable code segment, an expand-down data segment, or a stack segment. (This flag should always be set to 1 for 32-bit code and data segments and to 0 for 16-bit code and data segments.)

- **Executable code segment.** The flag is called the D flag and it indicates the default length for effective addresses and operands referenced by instructions in the segment. If the flag is set, 32-bit addresses and 32-bit or 8-bit operands are assumed; if it is clear, 16-bit addresses and 16-bit or 8-bit operands are assumed. The instruction prefix 66H can be used to select an operand size other than the default, and the prefix 67H can be used select an address size other than the default.

- **Stack segment (data segment pointed to by the SS register).** The flag is called the B (big) flag and it specifies the size of the stack pointer used for implicit stack operations (such as pushes, pops, and calls). If the flag is set, a 32-bit stack pointer is used, which is stored in the 32-bit ESP register; if the flag is clear, a 16-bit stack pointer is used, which is stored in the 16-bit SP register. If the stack segment is set up to be an expand-down data segment (described in the next paragraph), the B flag also specifies the upper bound of the stack segment.
- **Expand-down data segment.** The flag is called the B flag and it specifies the upper bound of the segment. If the flag is set, the upper bound is FFFFFFFFH (4 GBytes); if the flag is clear, the upper bound is FFFFH (64 KBytes).

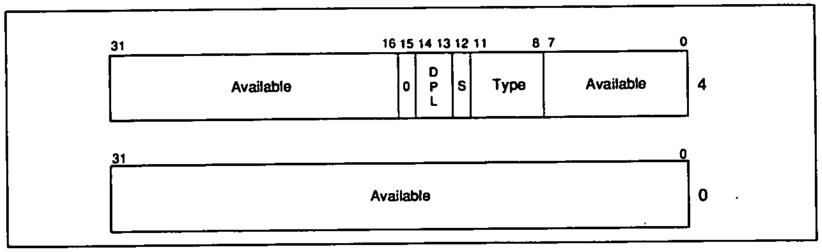


Figure 3-9. Segment Descriptor When Segment-Present Flag Is Clear

G (granularity) flag

Determines the scaling of the segment limit field. When the granularity flag is clear, the segment limit is interpreted in byte units; when flag is set, the segment limit is interpreted in 4-KByte units. (This flag does not affect the granularity of the base address; it is always byte granular.) When the granularity flag is set, the twelve least significant bits of an offset are not tested when checking the offset against the segment limit. For example, when the granularity flag is set, a limit of 0 results in valid offsets from 0 to 4095.

Available and reserved bits

Bit 20 of the second doubleword of the segment descriptor is available for use by system software; bit 21 is reserved and should always be set to 0.

3.4.3.1. CODE- AND DATA-SEGMENT DESCRIPTOR TYPES

When the S (descriptor type) flag in a segment descriptor is set, the descriptor is for either a code or a data segment. The highest order bit of the type field (bit 11 of the second double word of the segment descriptor) then determines whether the descriptor is for a data segment (clear) or a code segment (set).

For data segments, the three low-order bits of the type field (bits 8, 9, and 10) are interpreted as accessed (A), write-enable (W), and expansion-direction (E). See Table 3-1 for a description of the encoding of the bits in the type field for code and data segments. Data segments can be read-only or read/write segments, depending on the setting of the write-enable bit.

Table 3-1. Code- and Data-Segment Types

	Type Field					
Decimal	11	10 E	9 W	8 A	Descriptor Type	Description
0	0	0	0	0	Data	Read-Only
1	0	0	0	1	Data	Read-Only, accessed
2	0	0	1	0	Data	Read/Write
3	0	0	1	1	Data	Read/Write, accessed
4	0	1	0	0	Data	Read-Only, expand-down
5	0	1	0	1	Data	Read-Only, expand-down, accessed
6	0	1	1	0	Data	Read/Write, expand-down
7	0	1	1	1	Data	Read/Write, expand-down, accessed
		C	R	A		
8	1	0	0	0	Code	Execute-Only
9	1	0	0	1	Code	Execute-Only, accessed
10	1	0	1	0	Code	Execute/Read
11	1	0	1	1	Code	Execute/Read, accessed
12	1	1	0	0	Code	Execute-Only, conforming
13	1	1	0	1	Code	Execute-Only, conforming, accessed
14	1	1	1	0	Code	Execute/Read-Only, conforming
15	1	1	1	1	Code	Execute/Read-Only, conforming, accessed

Stack segments are data segments which must be read/write segments. Loading the SS register with a segment selector for a nonwritable data segment generates a general-protection exception (#GP). If the size of a stack segment needs to be changed dynamically, the stack segment can be an expand-down data segment (expansion-direction flag set). Here, dynamically changing the segment limit causes stack space to be added to the bottom of the stack. If the size of a stack segment is intended to remain static, the stack segment may be either an expand-up or expand-down type.

The accessed bit indicates whether the segment has been accessed since the last time the operating-system or executive cleared the bit. The processor sets this bit whenever it loads a segment selector for the segment into a segment register. The bit remains set until explicitly cleared. This bit can be used both for virtual memory management and for debugging.

For code segments, the three low-order bits of the type field are interpreted as accessed (A), read enable (R), and conforming (C). Code segments can be execute-only or execute/read, depending on the setting of the read-enable bit. An execute/read segment might be used when constants or other static data have been placed with instruction code in a ROM. Here, data can be read from

G

the code segment either by using an instruction with a CS override prefix or by loading a segment selector for the code segment in a data-segment register (the DS, ES, FS, or GS registers). In protected mode, code segments are not writable.

Code segments can be either conforming or nonconforming. A transfer of execution into a more-privileged conforming segment allows execution to continue at the current privilege level. A transfer into a nonconforming segment at a different privilege level results in a general-protection exception (#GP), unless a call gate or task gate is used (see Section 4.8.1., "Direct Calls or Jumps to Code Segments", for more information on conforming and nonconforming code segments). System utilities that do not access protected facilities and handlers for some types of exceptions (such as, divide error or overflow) may be loaded in conforming code segments. Utilities that need to be protected from less privileged programs and procedures should be placed in nonconforming code segments.

NOTE

> Execution cannot be transferred by a call or a jump to a less-privileged (numerically higher privilege level) code segment, regardless of whether the target segment is a conforming or nonconforming code segment. Attempting such an execution transfer will result in a general-protection exception.

All data segments are nonconforming, meaning that they cannot be accessed by less privileged programs or procedures (code executing at numerically high privilege levels). Unlike code segments, however, data segments can be accessed by more privileged programs or procedures (code executing at numerically lower privilege levels) without using a special access gate.

The processor may update the Type field when a segment is accessed, even if the access is a read cycle. If the descriptor tables have been put in ROM, it may be necessary for hardware to prevent the ROM from being enabled onto the data bus during a write cycle. It also may be necessary to return the READY# signal to the processor when a write cycle to ROM occurs, otherwise the cycle will not terminate. These features of the hardware design are necessary for using ROM-based descriptor tables with the Intel386 DX processor, which always sets the Accessed bit when a segment descriptor is loaded. The P6 family, Pentium, and Intel486 processors, however, only set the accessed bit if it is not already set. Writes to descriptor tables in ROM can be avoided by setting the accessed bits in every descriptor.

3.5. SYSTEM DESCRIPTOR TYPES

When the S (descriptor type) flag in a segment descriptor is clear, the descriptor type is a system descriptor. The processor recognizes the following types of system descriptors:

- Local descriptor-table (LDT) segment descriptor.
- Task-state segment (TSS) descriptor.
- Call-gate descriptor.
- Interrupt-gate descriptor.
- Trap-gate descriptor.

- Task-gate descriptor.

These descriptor types fall into two categories: system-segment descriptors and gate descriptors. System-segment descriptors point to system segments (LDT and TSS segments). Gate descriptors are in themselves "gates," which hold pointers to procedure entry points in code segments (call, interrupt, and trap gates) or which hold segment selectors for TSS's (task gates). Table 3-2 shows the encoding of the type field for system-segment descriptors and gate descriptors.

Table 3-2. System-Segment and Gate-Descriptor Types

Type Field					
Decimal	11	10	9	8	Description
0	0	0	0	0	Reserved
1	0	0	0	1	16-Bit TSS (Available)
2	0	0	1	0	LDT
3	0	0	1	1	16-Bit TSS (Busy)
4	0	1	0	0	16-Bit Call Gate
5	0	1	0	1	Task Gate
6	0	1	1	0	16-Bit Interrupt Gate
7	0	1	1	1	16-Bit Trap Gate
8	1	0	0	0	Reserved
9	1	0	0	1	32-Bit TSS (Available)
10	1	0	1	0	Reserved
11	1	0	1	1	32-Bit TSS (Busy)
12	1	1	0	0	32-Bit Call Gate
13	1	1	0	1	Reserved
14	1	1	1	0	32-Bit Interrupt Gate
15	1	1	1	1	32-Bit Trap Gate

For more information on the system-segment descriptors, see Section 3.5.1., "Segment Descriptor Tables", and Section 6.2.2., "TSS Descriptor"; for more information on the gate descriptors, see Section 4.8.2., "Gate Descriptors", Section 5.9., "IDT Descriptors", and Section 6.2.4., "Task-Gate Descriptor".

3.5.1. Segment Descriptor Tables

A segment descriptor table is an array of segment descriptors (see Figure 3-10). A descriptor table is variable in length and can contain up to 8192 (2^{13}) 8-byte descriptors. There are two kinds of descriptor tables:

- The global descriptor table (GDT)
- The local descriptor tables (LDT)

G

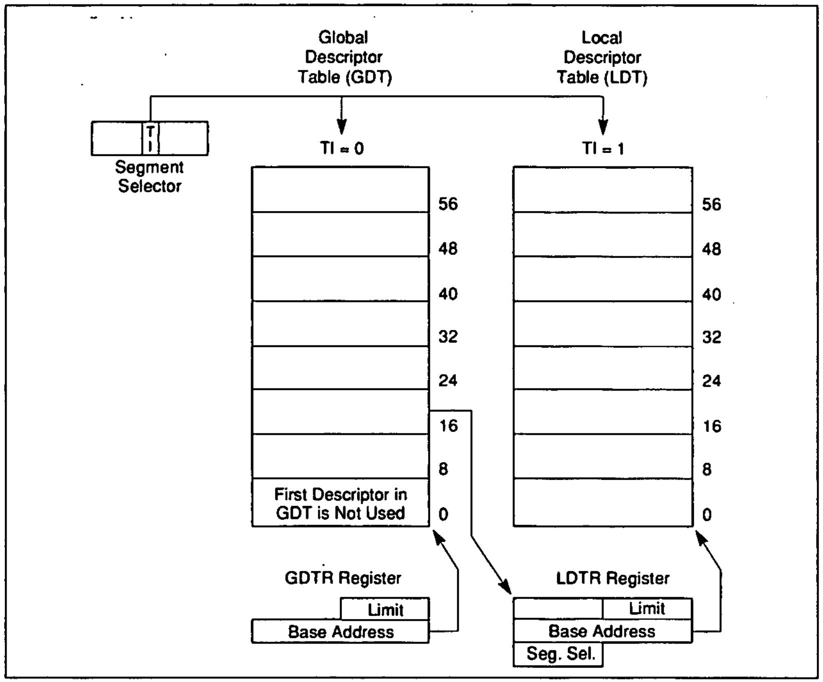


Figure 3-10. Global and Local Descriptor Tables

Each system must have one GDT defined, which may be used for all programs and tasks in the system. Optionally, one or more LDTs can be defined. For example, an LDT can be defined for each separate task being run, or some or all tasks can share the same LDT.

The GDT is not a segment itself; instead, it is a data structure in the linear address space. The base linear address and limit of the GDT must be loaded into the GDTR register (see Section 2.4., "Memory-Management Registers"). The base addresses of the GDT should be aligned on an eight-byte boundary to yield the best processor performance. The limit value for the GDT is expressed in bytes. As with segments, the limit value is added to the base address to get the address of the last valid byte. A limit value of 0 results in exactly one valid byte. Because segment descriptors are always 8 bytes long, the GDT limit should always be one less than an integral multiple of eight (that is, 8N – 1).

The first descriptor in the GDT is not used by the processor. A segment selector to this "null descriptor" does not generate an exception when loaded into a data-segment register (DS, ES, FS, or GS), but it always generates a general-protection exception (#GP) when an attempt is made to access memory using the descriptor. By initializing the segment registers with this segment selector, accidental reference to unused segment registers can be guaranteed to generate an exception.

The LDT is located in a system segment of the LDT type. The GDT must contain a segment descriptor for the LDT segment. If the system supports multiple LDTs, each must have a separate segment selector and segment descriptor in the GDT. The segment descriptor for an LDT can be located anywhere in the GDT. See Section 3.5., "System Descriptor Types", information on the LDT segment-descriptor type.

An LDT is accessed with its segment selector. To eliminate address translations when accessing the LDT, the segment selector, base linear address, limit, and access rights of the LDT are stored in the LDTR register (see Section 2.4., "Memory-Management Registers").

When the GDTR register is stored (using the SGDT instruction), a 48-bit "pseudo-descriptor" is stored in memory (see Figure 3-11). To avoid alignment check faults in user mode (privilege level 3), the pseudo-descriptor should be located at an odd word address (that is, address MOD 4 is equal to 2). This causes the processor to store an aligned word, followed by an aligned doubleword. User-mode programs normally do not store pseudo-descriptors, but the possibility of generating an alignment check fault can be avoided by aligning pseudo-descriptors in this way. The same alignment should be used when storing the IDTR register using the SIDT instruction. When storing the LDTR or task register (using the SLTR or STR instruction, respectively), the pseudo-descriptor should be located at a doubleword address (that is, address MOD 4 is equal to 0).

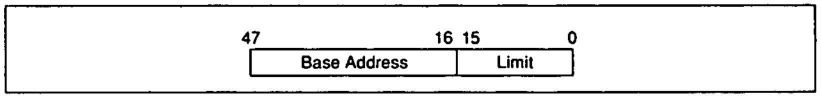


Figure 3-11. Pseudo-Descriptor Format

3.6. PAGING (VIRTUAL MEMORY)

When operating in protected mode, the Intel Architecture permits the linear address space to be mapped directly into a large physical memory (for example, 4 GBytes of RAM) or indirectly (using paging) into a smaller physical memory and disk storage. This latter method of mapping the linear address space is commonly referred to as virtual memory or demand-paged virtual memory.

When paging is used, the processor divides the linear address space into fixed-size pages (generally 4 KBytes in length) that can be mapped into physical memory and/or disk storage. When a program (or task) references a logical address in memory, the processor translates the address into a linear address and then uses its paging mechanism to translate the linear address into a corresponding physical address. If the page containing the linear address is not currently in physical memory, the processor generates a page-fault exception (#PF). The exception handler for the page-fault exception typically directs the operating system or executive to load the page from disk storage into physical memory (perhaps writing a different page from physical memory out to disk in the process). When the page has been loaded in physical memory, a return from the exception handler causes the instruction that generated the exception to be restarted. The information that the processor uses to map linear addresses into the physical address space and

9

to generate page-fault exceptions (when necessary) is contained in page directories and page tables stored in memory.

Paging is different from segmentation through its use of fixed-size pages. Unlike segments, which usually are the same size as the code or data structures they hold, pages have a fixed size. If segmentation is the only form of address translation used, a data structure present in physical memory will have all of its parts in memory. If paging is used, a data structure can be partly in memory and partly in disk storage.

To minimize the number of bus cycles required for address translation, the most recently accessed page-directory and page-table entries are cached in the processor in devices called translation lookaside buffers (TLBs). The TLBs satisfy most requests for reading the current page directory and page tables without requiring a bus cycle. Extra bus cycles occur only when the TLBs do not contain a page-table entry, which typically happens when a page has not been accessed for a long time. See Section 3.7., "Translation Lookaside Buffers (TLBs)", for more information on the TLBs.

3.6.1. Paging Options

Paging is controlled by three flags in the processor's control registers:

- PG (paging) flag, bit 31 of CR0 (available in all Intel Architecture processors beginning with the Intel386™ processor).
- PSE (page size extensions) flag, bit 4 of CR4 (introduced in the Pentium® and Pentium Pro processors).
- PAE (physical address extension) flag, bit 5 of CR4 (introduced in the Pentium Pro processors).

The PG flag enables the page-translation mechanism. The operating system or executive usually sets this flag during processor initialization. The PG flag must be set if the processor's page-translation mechanism is to be used to implement a demand-paged virtual memory system or if the operating system is designed to run more than one program (or task) in virtual-8086 mode.

The PSE flag enables large page sizes: 4-MByte pages or 2-MByte pages (when the PAE flag is set). When the PSE flag is clear, the more common page length of 4 KBytes is used. See Section 3.6.2.2., "Linear Address Translation (4-MByte Pages)", and Section 3.8.2., "Linear Address Translation With Extended Addressing Enabled (2-MByte Pages)", for more information about the use of the PSE flag.

The PAE flag enables 36-bit physical addresses. This physical address extension can only be used when paging is enabled. It relies on page directories and page tables to reference physical addresses above FFFFFFFFH. See Section 3.8., "Physical Address Extension", for more information about the physical address extension.

3.6.2. Page Tables and Directories

The information that the processor uses to translate linear addresses into physical addresses (when paging is enabled) is contained in four data structures:

- Page directory—An array of 32-bit page-directory entries (PDEs) contained in a 4-KByte page. Up to 1024 page-directory entries can be held in a page directory.
- Page table—An array of 32-bit page-table entries (PTEs) contained in a 4-KByte page. Up to 1024 page-table entries can be held in a page table. (Page tables are not used for 2-MByte or 4-MByte pages. These page sizes are mapped directly from one or more page-directory entries.)
- Page—A 4-KByte, 2-MByte, or 4-MByte flat address space.
- Page-Directory-Pointer Table—An array of four 64-bit entries, each of which points to a page directory. This data structure is only used when the physical address extension is enabled (see Section 3.8., "Physical Address Extension").

These tables provide access to either 4-KByte or 4-MByte pages when normal 32-bit physical addressing is being used and to either 4-KByte or 2-MByte pages when extended (36-bit) physical addressing is being used. Table 3-3 shows the page size and physical address size obtained from various settings of the paging control flags. Each page-directory entry contains a PS (page size) flag that specifies whether the entry points to a page table whose entries in turn point to 4-KByte pages (PS set to 0) or whether the page-directory entry points directly to a 4-MByte or 2-MByte page (PSE or PAE set to 1 and PS set to 1).

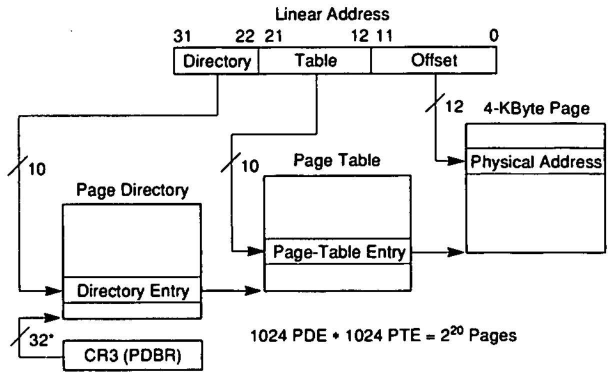
Table 3-3. Page Sizes and Physical Address Sizes

PG Flag, CR0	PAE Flag, CR4	PSE Flag, CR4	PS Flag, PDE	Page Size	Physical Address Size
0	X	X	X	—	Paging Disabled
1	0	0	X	4 KBytes	32 Bits
1	0	1	0	4 KBytes	32 Bits
1	0	1	1	4 MBytes	32 Bits
1	1	X	0	4 KBytes	36 Bits
1	1	X	1	2 MBytes	36 Bits

3.6.2.1. LINEAR ADDRESS TRANSLATION (4-KBYTE PAGES)

Figure 3-12 shows the page directory and page-table hierarchy when mapping linear addresses to 4-KByte pages. The entries in the page directory point to page tables, and the entries in a page table point to pages in physical memory. This paging method can be used to address up to 2^{20} pages, which spans a linear address space of 2^{32} bytes (4 GBytes).

9



*32 bits aligned onto a 4-KByte boundary.

Figure 3-12. Linear Address Translation (4-KByte Pages)

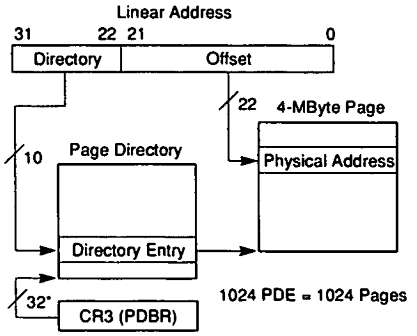
To select the various table entries, the linear address is divided into three sections:

- Page-directory entry—Bits 22 through 31 provide an offset to an entry in the page directory. The selected entry provides the base physical address of a page table.
- Page-table entry—Bits 12 through 21 of the linear address provide an offset to an entry in the selected page table. This entry provides the base physical address of a page in physical memory.
- Page offset—Bits 0 through 11 provides an offset to a physical address in the page.

Memory management software has the option of using one page directory for all programs and tasks, one page directory for each task, or some combination of the two.

3.6.2.2. LINEAR ADDRESS TRANSLATION (4-MBYTE PAGES)

Figure 3-12 shows how a page directory can be used to map linear addresses to 4-MByte pages. The entries in the page directory point to 4-MByte pages in physical memory. This paging method can be used to map up to 1024 pages into a 4-GByte linear address space.



*32 bits aligned onto a 4-KByte boundary.

Figure 3-13. Linear Address Translation (4-MByte Pages)

The 4-MByte page size is selected by setting the PSE flag in control register CR4 and setting the page size (PS) flag in a page-directory entry (see Figure 3-14). With these flags set, the linear address is divided into two sections:

- Page directory entry—Bits 22 through 31 provide an offset to an entry in the page directory. The selected entry provides the base physical address of a 4-MByte page.
- Page offset—Bits 0 through 21 provides an offset to a physical address in the page.

NOTE

> (For the Pentium® processor only.) When enabling or disabling large page sizes, the TLBs must be invalidated (flushed) after the PSE flag in control register CR4 has been set or cleared. Otherwise, incorrect page translation might occur due to the processor using outdated page translation information stored in the TLBs. See Section 9.9., "Invalidating the Translation Lookaside Buffers (TLBs)", for information on how to invalidate the TLBs.

3.6.2.3. MIXING 4-KBYTE AND 4-MBYTE PAGES

When the PSE flag in CR4 is set, both 4-MByte pages and page tables for 4-KByte pages can be accessed from the same page directory. If the PSE flag is clear, only page tables for 4-KByte pages can be accessed (regardless of the setting of the PS flag in a page-directory entry).

A typical example of mixing 4-KByte and 4-MByte pages is to place the operating system or executive's kernel in a large page to reduce TLB misses and thus improve overall system performance. The processor maintains 4-MByte page entries and 4-KByte page entries in separate

9

TLBs. So, placing often used code such as the kernel in a large page, frees up 4-KByte-page TLB entries for application programs and tasks.

3.6.3. Base Address of the Page Directory

The physical address of the current page directory is stored in the CR3 register (also called the page directory base register or PDBR). (See Figure 2-5 and Section 2.5., "Control Registers", for more information on the PDBR.) If paging is to be used, the PDBR must be loaded as part of the processor initialization process (prior to enabling paging). The PDBR can then be changed either explicitly by loading a new value in CR3 with a MOV instruction or implicitly as part of a task switch. (See Section 6.2.1., "Task-State Segment (TSS)", for a description of how the contents of the CR3 register is set for a task.)

There is no present flag in the PDBR for the page directory. The page directory may be not-present (paged out of physical memory) while its associated task is suspended, but the operating system must ensure that the page directory indicated by the PDBR image in a task's TSS is present in physical memory before the task is dispatched. The page directory must also remain in memory as long as the task is active.

3.6.4. Page-Directory and Page-Table Entries

Figure 3-14 shows the format for the page-directory and page-table entries when 4-KByte pages and 32-bit physical addresses are being used. Figure 3-14 shows the format for the page-directory entries when 4-MByte pages and 32-bit physical addresses are being used. See Section 3.8., "Physical Address Extension", for the format of page-directory and page-table entries when the physical address extension is being used.

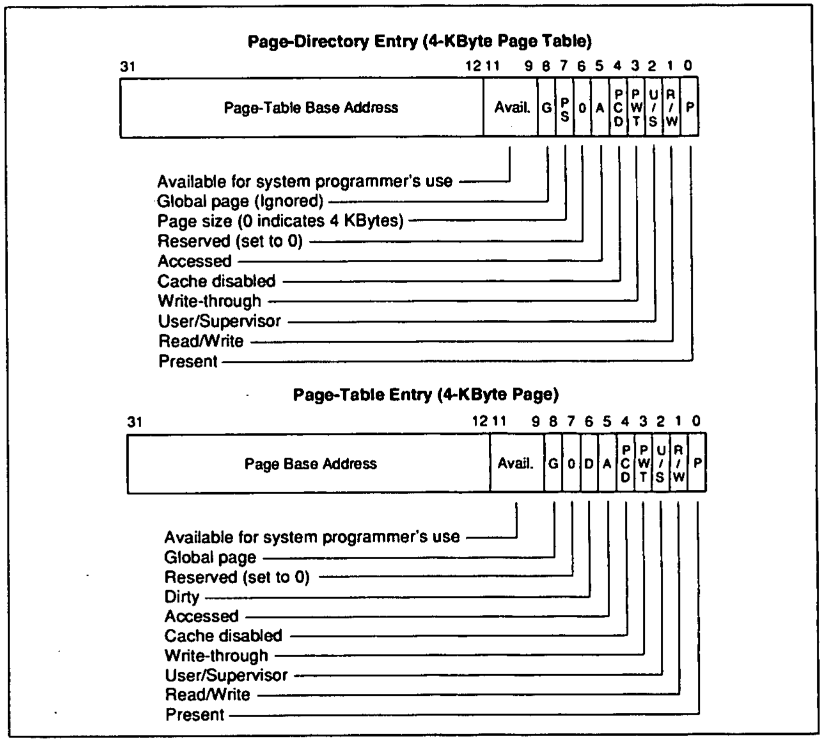


Figure 3-14. Format of Page-Directory and Page-Table Entries for 4-KByte Pages and 32-Bit Physical Addresses

9

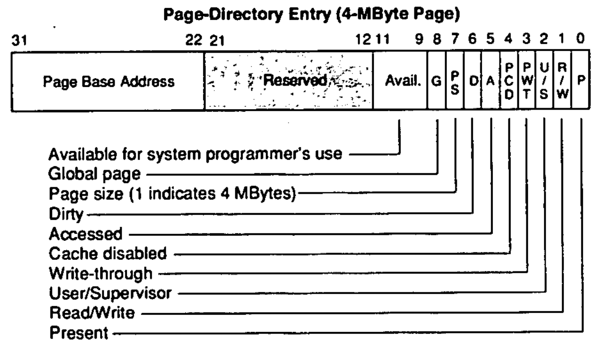


Figure 3-15. Format of Page-Directory Entries for 4-MByte Pages and 32-Bit Addresses

The functions of the flags and fields in the entries in Figures 3-14 and 3-15 are as follows:

Page base address, bits 12 through 32

(Page-table entries for 4-KByte pages.) Specifies the physical address of the first byte of a 4-KByte page. The bits in this field are interpreted as the 20 most-significant bits of the physical address, which forces pages to be aligned on 4-KByte boundaries.

(Page-directory entries for 4-KByte page tables.) Specifies the physical address of the first byte of a page table. The bits in this field are interpreted as the 20 most-significant bits of the physical address, which forces page tables to be aligned on 4-KByte boundaries.

(Page-directory entries for 4-MByte pages.) Specifies the physical address of the first byte of a 4-MByte page. Only bits 22 through 31 of this field are used (and bits 12 through 21 are reserved and must be set to 0, for Intel Architecture processors through the Pentium® II processor). The base address bits are interpreted as the 10 most-significant bits of the physical address, which forces 4-MByte pages to be aligned on 4-MByte boundaries.

Present (P) flag, bit 0

Indicates whether the page or page table being pointed to by the entry is currently loaded in physical memory. When the flag is set, the page is in physical memory and address translation is carried out. When the flag is clear, the page is not in memory and, if the processor attempts to access the page, it generates a page-fault exception (#PF).

The processor does not set or clear this flag; it is up to the operating system or executive to maintain the state of the flag.

If the processor generates a page-fault exception, the operating system generally needs to carry out the following operations:

1. Copy the page from disk storage into physical memory.
2. Load the page address into the page-table or page-directory entry and set its present flag. Other flags, such as the dirty and accessed flags, may also be set at this time.
3. Invalidate the current page-table entry in the TLB (see Section 3.7., "Translation Lookaside Buffers (TLBs)", for a discussion of TLBs and how to invalidate them).
4. Return from the page-fault handler to restart the interrupted program (or task).

Read/write (R/W) flag, bit 1

Specifies the read-write privileges for a page or group of pages (in the case of a page-directory entry that points to a page table). When this flag is clear, the page is read only; when the flag is set, the page can be read and written into. This flag interacts with the U/S flag and the WP flag in register CR0. See Section 4.11., "Page-Level Protection", and Table 4-2 for a detailed discussion of the use of these flags.

User/supervisor (U/S) flag, bit 2

Specifies the user-supervisor privileges for a page or group of pages (in the case of a page-directory entry that points to a page table). When this flag is clear, the page is assigned the supervisor privilege level; when the flag is set, the page is assigned the user privilege level. This flag interacts with the R/W flag and the WP flag in register CR0. See Section 4.11., "Page-Level Protection", and Table 4-2 for a detail discussion of the use of these flags.

Page-level write-through (PWT) flag, bit 3

Controls the write-through or write-back caching policy of individual pages or page tables. When the PWT flag is set, write-through caching is enabled for the associated page or page table; when the flag is clear, write-back caching is enabled for the associated page or page table. The processor ignores this flag if the CD (cache disable) flag in CR0 is set. See Section 9.5., "Cache Control", for more information about the use of this flag. See Section 2.5., "Control Registers", for a description of a companion PWT flag in control register CR3.

Page-level cache disable (PCD) flag, bit 4

Controls the caching of individual pages or page tables. When the PCD flag is set, caching of the associated page or page table is prevented; when the flag is clear, the page or page table can be cached. This flag permits caching to be disabled for pages that contain memory-mapped I/O ports or that do not provide a performance benefit when cached. The processor ignores this flag (assumes it is set) if the CD (cache disable) flag in CR0 is set. See Chapter 9, *Memory Cache Control*, for more information about the use of this flag. See Section 2.5., "Control Registers", for a description of a companion PCD flag in control register CR3.

9

Accessed (A) flag, bit 5

Indicates whether a page or page table has been accessed (read from or written to) when set. Memory management software typically clears this flag when a page or page table is initially loaded into physical memory. The processor then sets this flag the first time a page or page table is accessed. This flag is a "sticky" flag, meaning that once set, the processor does not implicitly clear it. Only software can clear this flag. The accessed and dirty flags are provided for use by memory management software to manage the transfer of pages and page tables into and out of physical memory.

Dirty (D) flag, bit 6

Indicates whether a page has been written to when set. (This flag is not used in page-directory entries that point to page tables.) Memory management software typically clears this flag when a page is initially loaded into physical memory. The processor then sets this flag the first time a page is accessed for a write operation. This flag is "sticky," meaning that once set, the processor does not implicitly clear it. Only software can clear this flag. The dirty and accessed flags are provided for use by memory management software to manage the transfer of pages and page tables into and out of physical memory.

Page size (PS) flag, bit 7

Determines the page size. This flag is only used in page-directory entries. When this flag is clear, the page size is 4 KBytes and the page-directory entry points to a page table. When the flag is set, the page size is 4 MBytes for normal 32-bit addressing (and 2 MBytes if extended physical addressing is enabled) and the page-directory entry points to a page. If the page-directory entry points to a page table, all the pages associated with that page table will be 4-KByte pages.

Global (G) flag, bit 8

(Introduced in the Pentium Pro processor.) Indicates a global page when set. When a page is marked global and the page global enable (PGE) flag in register CR4 is set, the page-table or page-directory entry for the page is not invalidated in the TLB when register CR3 is loaded or a task switch occurs. This flag is provided to prevent frequently used pages (such as pages that contain kernel or other operating system or executive code) from being flushed from the TLB. Only software can set or clear this flag. For page-directory entries that point to page tables, this flag is ignored and the global characteristics of a page are set in the page-table entries. See Section 3.7., "Translation Lookaside Buffers (TLBs)", for more information about the use of this flag. (This bit is reserved in Pentium and earlier Intel Architecture processors.)

Reserved and available-to-software bits

In a page-table entry, bit 7 is reserved and should be set to 0; in a page-directory entry that points to a page table, bit 6 is reserved and should be set to 0. For a page-directory entry for a 4-MByte page, bits 12 through 21 are reserved and must be set to 0, for Intel Architecture processors through the Pentium II processor. For both types of entries, bits 9, 10, and 11 are available for use by software. (When the present bit is clear, bits 1 through 31 are available to software—see Figure 3-16.) When the PSE and PAE flags in control register CR4 are set, the processor generates a page fault if reserved bits are not set to 0.

3.6.5. Not Present Page-Directory and Page-Table Entries

When the present flag is clear for a page-table or page-directory entry, the operating system or executive may use the rest of the entry for storage of information such as the location of the page in the disk storage system (see Figure 3-16).

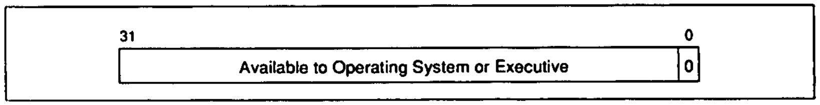


Figure 3-16. Format of a Page-Table or Page-Directory Entry for a Not-Present Page

3.7. TRANSLATION LOOKASIDE BUFFERS (TLBS)

The processor stores the most recently used page-directory and page-table entries in on-chip caches called translation lookaside buffers or TLBs. The P6 family and Pentium processors have separate TLBs for the data and instruction caches. Also, the P6 family processors maintain separate TLBs for 4-KByte and 4-MByte page sizes. The CPUID instruction can be used to determine the sizes of the TLBs provided in the P6 family and Pentium processors.

Most paging is performed using the contents of the TLBs. Bus cycles to the page directory and page tables in memory are performed only when the TLBs do not contain the translation information for a requested page.

The TLBs are inaccessible to application programs and tasks (privilege level greater than 0); that is, they cannot invalidate TLBs. Only, operating system or executive procedures running at privilege level of 0 can invalid TLBs or selected TBL entries. Whenever a page-directory or page-table entry is changed (including when the present flag is set to zero), the operating-system must immediately invalidate the corresponding entry in the TLB so that it can be updated the next time the entry is referenced.

All of the (nonglobal) TLBs are automatically invalidated any time the CR3 register is loaded (unless the G flag for a page or page-table entry is set, as describe later in this section). The CR3 register can be loaded in either of two ways:

- Explicitly, using the MOV instruction, for example:

  ```
  MOV CR3, EAX
  ```

  where the EAX register contains an appropriate page-directory base address.

- Implicitly by executing a task switch, which automatically changes the contents of the CR3 register.

The INVLPG instruction is provided to invalidate a specific page-table entry in the TLB. Normally, this instruction invalidates only an individual TLB entry; however, in some cases, it

9

may invalidate more than the selected entry and may even invalidate all of the TLBs. This instruction ignores the setting of the G flag in a page-directory or page-table entry (see following paragraph).

(Introduced in the Pentium Pro processor.) The page global enable (PGE) flag in register CR4 and the global (G) flag of a page-directory or page-table entry (bit 8) can be used to prevent frequently used pages from being automatically invalidated in the TLBs on a task switch or a load of register CR3. (See Section 3.6.4., "Page-Directory and Page-Table Entries", for more information about the global flag.) When the processor loads a page-directory or page-table entry for a global page into a TLB, the entry will remain in the TLB indefinitely. The only way to deterministically invalidate global page entries is to clear the PGE flag and then invalidate the TLBs or to use the INVLPG instruction to invalidate individual page-directory or page-table entries in the TLBs.

For additional information about invalidation of the TLBs, see Section 9.9., "Invalidating the Translation Lookaside Buffers (TLBs)".

3.8. PHYSICAL ADDRESS EXTENSION

The physical address extension (PAE) flag in register CR4 enables an extension of physical addresses from 32 bits to 36 bits. (This feature was introduced into the Intel Architecture in the Pentium Pro processors.) Here, the processor provides 4 additional address line pins to accommodate the additional address bits. This option can only be used when paging is enabled (that is, when both the PG flag in register CR0 and the PAE flag in register CR4 are set).

When the physical address extension is enabled, the processor allows two sizes of pages: 4-KByte and 2-MByte. As with 32-bit addressing, both page sizes can be addressed within the same set of paging tables (that is, a page-directory entry can point to either a 2-MByte page or a page table that in turn points to 4-KByte pages). To support the 36-bit physical addresses, the following changes are made to the paging data structures:

- The paging table entries are increased to 64 bits to accommodate 36-bit base physical addresses. Each 4-KByte page directory and page table can thus have up to 512 entries.
- A new table, called the page-directory-pointer table, is added to the linear-address translation hierarchy. This table has 4 entries of 64-bits each, and it lies above the page directory in the hierarchy. With the physical address extension mechanism enabled, the processor supports up to 4 page directories.
- The 20-bit page-directory base address field in register CR3 (PDPR) is replaced with a 27-bit page-directory-pointer-table base address field (see Figure 3-17). (In this case, register CR3 is called the PDPTR.) This field provides the 27 most-significant bits of the physical address of the first byte of the page-directory-pointer table, which forces the table to be located on a 32-byte boundary.
- Linear address translation is changed to allow mapping 32-bit linear addresses into the larger physical address space.

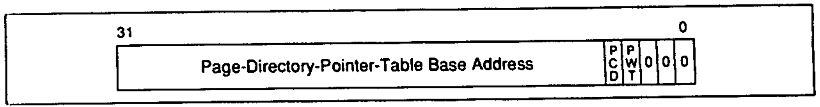


Figure 3-17. Register CR3 Format When the Physical Address Extension Is Enabled

3.8.1. Linear Address Translation With Extended Addressing Enabled (4-KByte Pages)

Figure 3-12 shows the page-directory-pointer, page-directory, and page-table hierarchy when mapping linear addresses to 4-KByte pages with extended physical addressing enabled. This paging method can be used to address up to 2^{20} pages, which spans a linear address space of 2^{32} bytes (4 GBytes).

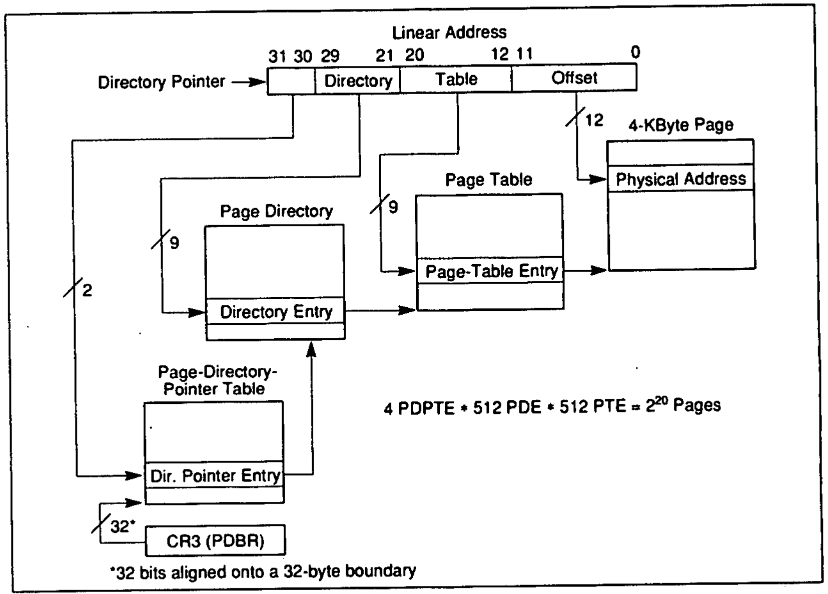


Figure 3-18. Linear Address Translation With Extended Physical Addressing Enabled (4-KByte Pages)

To select the various table entries, the linear address is divided into three sections:

- Page-directory-pointer-table entry—Bits 30 and 31 provide an offset to one of the 4 entries in the page-directory-pointer table. The selected entry provides the base physical address of a page directory.
- Page-directory entry—Bits 21 through 29 provide an offset to an entry in the selected page directory. The selected entry provides the base physical address of a page table.
- Page-table entry—Bits 12 through 20 provide an offset to an entry in the selected page table. This entry provides the base physical address of a page in physical memory.
- Page offset—Bits 0 through 11 provide an offset to a physical address in the page.

3.8.2. Linear Address Translation With Extended Addressing Enabled (2-MByte Pages)

Figure 3-12 shows how a page-directory-pointer table and page directories can be used to map linear addresses to 2-MByte pages. This paging method can be used to map up to 2048 pages (4 page-directory-pointer-table entries times 512 page-directory entries) into a 4-GByte linear address space.

The 2-MByte page size is selected by setting the PSE flag in control register CR4 and setting the page size (PS) flag in a page-directory entry (see Figure 3-14). With these flags set, the linear address is divided into three sections:

- Page-directory-pointer-table entry—Bits 30 and 31 provide an offset to an entry in the page-directory-pointer table. The selected entry provides the base physical address of a page directory.
- Page-directory entry—Bits 21 through 29 provide an offset to an entry in the page directory. The selected entry provides the base physical address of a 2-MByte page.
- Page offset—Bits 0 through 20 provides an offset to a physical address in the page.

3.8.3. Accessing the Full Extended Physical Address Space With the Extended Page-Table Structure

The page-table structure described in the previous two sections allows up to 4 GBytes of the 64 GByte extended physical address space to be addressed at one time. Additional 4-GByte sections of physical memory can be addressed in either of two way:

- Change the pointer in register CR3 to point to another page-directory-pointer table, which in turn points to another set of page directories and page tables.
- Change entries in the page-directory-pointer table to point to other page directories, which in turn point to other sets of page tables.

G

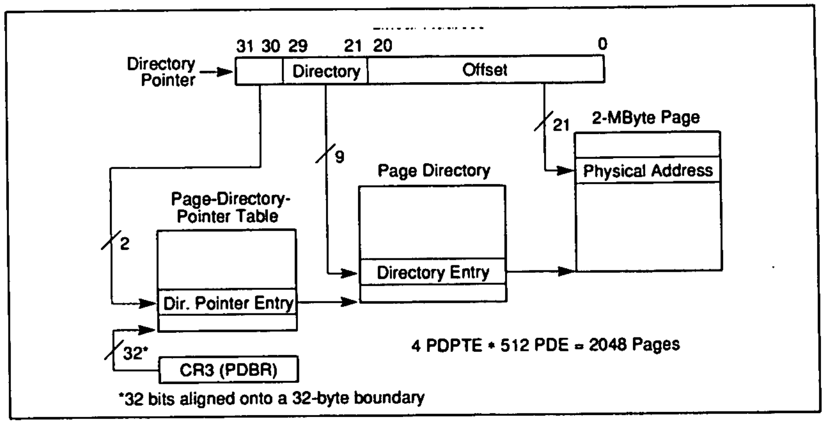


Figure 3-19. Linear Address Translation With Extended Physical Addressing Enabled (2-MByte Pages)

3.8.4. Page-Directory and Page-Table Entries With Extended Addressing Enabled

Figure 3-20 shows the format for the page-directory-pointer-table, page-directory, and page-table entries when 4-KByte pages and 36-bit extended physical addresses are being used. Figure 3-21 shows the format for the page-directory-pointer-table and page-directory entries when 2-MByte pages and 36-bit extended physical addresses are being used. The functions of the flags in these entries are the same as described in Section 3.6.4., "Page-Directory and Page-Table Entries". The major differences in these entries are as follows:

- A page-directory-pointer-table entry is added.
- The size of the entries are increased from 32 bits to 64 bits.
- The maximum number of entries in a page directory or page table is 512.
- The base physical address field in each entry is extended to 24 bits.

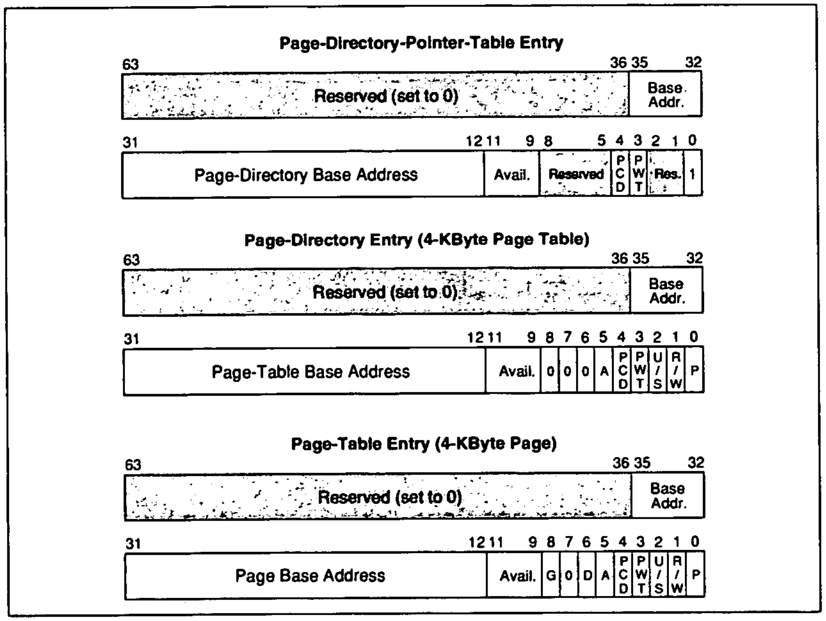


Figure 3-20. Format of Page-Directory-Pointer-Table, Page-Directory, and Page-Table Entries for 4-KByte Pages and 36-Bit Extended Physical Addresses

The base physical address in an entry specifies the following, depending on the type of entry:

- Page-directory-pointer-table entry—the physical address of the first byte of a 4-KByte page directory.
- Page-directory entry—the physical address of the first byte of a 4-KByte page table or a 2-MByte page.
- Page-table entry—the physical address of the first byte of a 4-KByte page.

For all table entries (except for page-directory entries that point to 2-MByte pages), the bits in the page base address are interpreted as the 24 most-significant bits of a 36-bit physical address, which forces page tables and pages to be aligned on 4-KByte boundaries. When a page-directory entry points to a 2-MByte page, the base address is interpreted as the 15 most-significant bits of a 36-bit physical address, which forces pages to be aligned on 2-MByte boundaries.

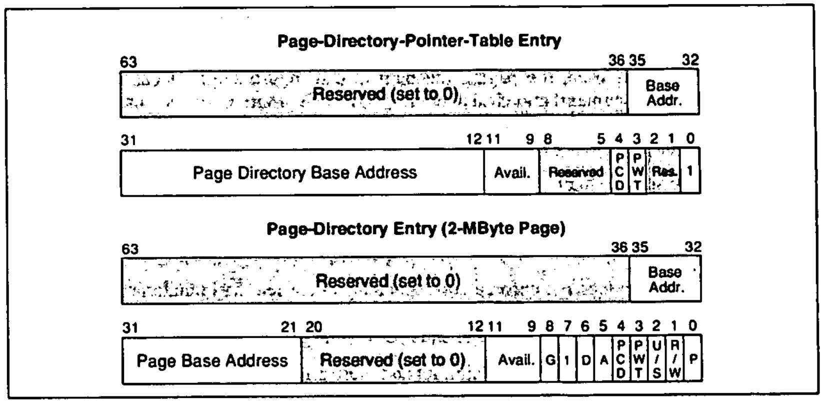


Figure 3-21. Format of Page-Directory-Pointer-Table and Page-Directory Entries for 2-MByte Pages and 36-Bit Extended Physical Addresses

The present flag (bit 0) in all page-directory-pointer-table entries must be set to 1 anytime extended physical addressing mode is enabled; that is, whenever the PAE flag (bit 5 in register CR4) and the PG flag (bit 31 in register CR0) are set. If the P flag is not set in all 4 page-directory-pointer-table entries in the page-directory-pointer table when extended physical addressing is enabled, a general-protection exception (#GP) is generated.

The page size (PS) flag (bit 7) in a page-directory entry determines if the entry points to a page table or a 2-MByte page. When this flag is clear, the entry points to a page table; when the flag is set, the entry points to a 2-MByte page. This flag allows 4-KByte and 2-MByte pages to be mixed within one set of paging tables.

Access (A) and dirty (D) flags (bits 5 and 6) are provided for table entries that point to pages.

Bits 9, 10, and 11 in all the table entries for the physical address extension are available for use by software. (When the present flag is clear, bits 1 through 63 are available to software.) All bits in Figure 3-14 that are marked reserved or 0 should be set to 0 by software and not accessed by software. When the PSE and/or PAE flags in control register CR4 are set, the processor generates a page fault (#PF) if reserved bits in page-directory and page-table entries are not set to 0, and it generates a general-protection exception (#GP) if reserved bits in a page-directory-pointer-table entry are not set to 0.

3.9. MAPPING SEGMENTS TO PAGES

The segmentation and paging mechanisms provide in the Intel Architecture support a wide variety of approaches to memory management. When segmentation and paging is combined, segments can be mapped to pages in several ways. To implement a flat (unsegmented) addressing environment, for example, all the code, data, and stack modules can be mapped to one or more large segments (up to 4-GBytes) that share same range of linear addresses (see Figure 3-2). Here, segments are essentially invisible to applications and the operating-system or executive. If paging is used, the paging mechanism can map a single linear address space (contained in a single segment) into virtual memory. Or, each program (or task) can have its own large linear address space (contained in its own segment), which is mapped into virtual memory through its own page directory and set of page tables.

Segments can be smaller than the size of a page. If one of these segments is placed in a page which is not shared with another segment, the extra memory is wasted. For example, a small data structure, such as a 1-byte semaphore, occupies 4K bytes if it is placed in a page by itself. If many semaphores are used, it is more efficient to pack them into a single page.

The Intel Architecture does not enforce correspondence between the boundaries of pages and segments. A page can contain the end of one segment and the beginning of another. Likewise, a segment can contain the end of one page and the beginning of another.

Memory-management software may be simpler and more efficient if it enforces some alignment between page and segment boundaries. For example, if a segment which can fit in one page is placed in two pages, there may be twice as much paging overhead to support access to that segment.

One approach to combining paging and segmentation that simplifies memory-management software is to give each segment its own page table, as shown in Figure 3-22. This convention gives the segment a single entry in the page directory which provides the access control information for paging the entire segment.

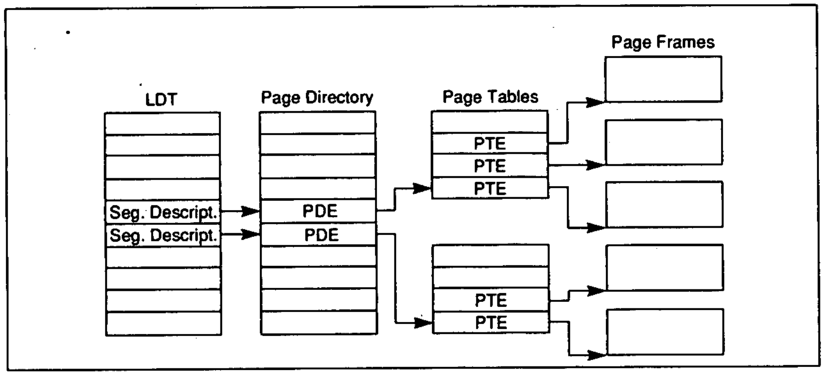


Figure 3-22. Memory Management Convention That Assigns a Page Table to Each Segment

4

Protection

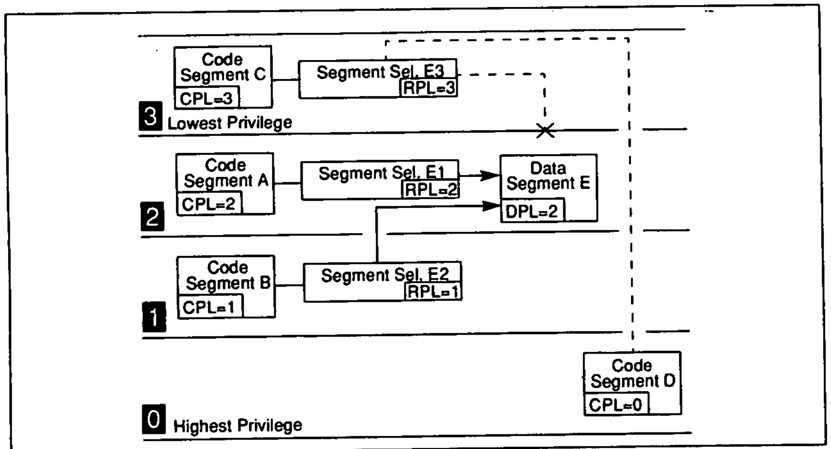


Figure 4-4. Examples of Accessing Data Segments From Various Privilege Levels

- The procedure in code segment D should be able to access data segment E because code segment D's CPL is numerically less than the DPL of data segment E. However, the RPL of segment selector E3 (which the code segment D procedure is using to access data segment E) is numerically greater than the DPL of data segment E, so access is not allowed. If the code segment D procedure were to use segment selector E1 or E2 to access the data segment, access would be allowed.

As demonstrated in the previous examples, the addressable domain of a program or task varies as its CPL changes. When the CPL is 0, data segments at all privilege levels are accessible; when the CPL is 1, only data segments at privilege levels 1 through 3 are accessible; when the CPL is 3, only data segments at privilege level 3 are accessible.

The RPL of a segment selector can always override the addressable domain of a program or task. When properly used, RPLs can prevent problems caused by accidental (or intensional) use of segment selectors for privileged data segments by less privileged programs or procedures.

It is important to note that the RPL of a segment selector for a data segment is under software control. For example, an application program running at a CPL of 3 can set the RPL for a data-segment selector to 0. With the RPL set to 0, only the CPL checks, not the RPL checks, will provide protection against deliberate, direct attempts to violate privilege-level security for the data segment. To prevent these types of privilege-level-check violations, a program or procedure can check access privileges whenever it receives a data-segment selector from another procedure (see Section 4.10.4., "Checking Caller Access Privileges (ARPL Instruction)").

4.6.1. Accessing Data in Code Segments

In some instances it may be desirable to access data structures that are contained in a code segment. The following methods of accessing data in code segments are possible:

- Load a data-segment register with a segment selector for a nonconforming, readable, code segment.
- Load a data-segment register with a segment selector for a conforming, readable, code segment.
- Use a code-segment override prefix (CS) to read a readable, code segment whose selector is already loaded in the CS register.

The same rules for accessing data segments apply to method 1. Method 2 is always valid because the privilege level of a conforming code segment is effectively the same as the CPL, regardless of its DPL. Method 3 is always valid because the DPL of the code segment selected by the CS register is the same as the CPL.

4.7. PRIVILEGE LEVEL CHECKING WHEN LOADING THE SS REGISTER

Privilege level checking also occurs when the SS register is loaded with the segment selector for a stack segment. Here all privilege levels related to the stack segment must match the CPL; that is, the CPL, the RPL of the stack-segment selector, and the DPL of the stack-segment descriptor must be the same. If the RPL and DPL are not equal to the CPL, a general-protection exception (#GP) is generated.

4.8. PRIVILEGE LEVEL CHECKING WHEN TRANSFERRING PROGRAM CONTROL BETWEEN CODE SEGMENTS

To transfer program control from one code segment to another, the segment selector for the destination code segment must be loaded into the code-segment register (CS). As part of this loading process, the processor examines the segment descriptor for the destination code segment and performs various limit, type, and privilege checks. If these checks are successful, the CS register is loaded, program control is transferred to the new code segment, and program execution begins at the instruction pointed to by the EIP register.

Program control transfers are carried out with the JMP, CALL, RET, INT *n*, and IRET instructions, as well as by the exception and interrupt mechanisms. Exceptions, interrupts, and the IRET instruction are special cases discussed in Chapter 5, *Interrupt and Exception Handling*. This chapter discusses only the JMP, CALL, and RET instructions.

A JMP or CALL instruction can reference another code segment in any of four ways:

- The target operand contains the segment selector for the target code segment.
- The target operand points to a call-gate descriptor, which contains the segment selector for the target code segment.

- The target operand points to a TSS, which contains the segment selector for the target code segment.
- The target operand points to a task gate, which points to a TSS, which in turn contains the segment selector for the target code segment.

The following sections describe first two types of references. See Section 6.3., "Task Switching", for information on transferring program control through a task gate and/or TSS.

4.8.1. Direct Calls or Jumps to Code Segments

The near forms of the JMP, CALL, and RET instructions transfer program control within the current code segment, so privilege-level checks are not performed. The far forms of the JMP, CALL, and RET instructions transfer control to other code segments, so the processor does perform privilege-level checks.

When transferring program control to another code segment without going through a call gate, the processor examines four kinds of privilege level and type information (see Figure 4-5):

- The CPL. (Here, the CPL is the privilege level of the calling code segment; that is, the code segment that contains the procedure that is making the call or jump.)

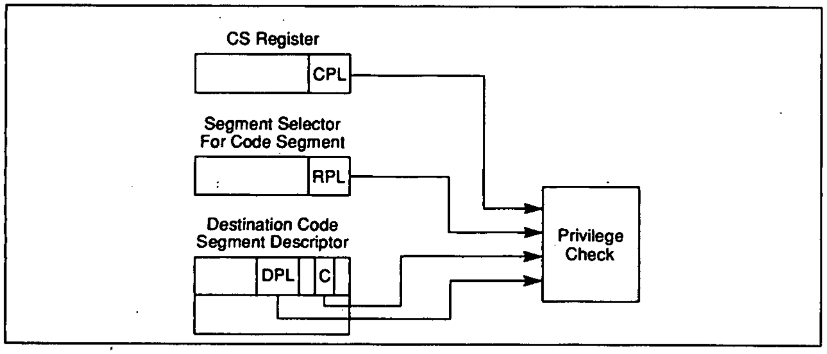

CS Register
CPL
Segment Selector For Code Segment
RPL
Destination Code Segment Descriptor
DPL
C
Privilege Check

Figure 4-5. Privilege Check for Control Transfer Without Using a Gate

- The DPL of the segment descriptor for the destination code segment that contains the called procedure.
- The RPL of the segment selector of the destination code segment.
- The conforming (C) flag in the segment descriptor for the destination code segment, which determines whether the segment is a conforming (C flag is set) or nonconforming (C flag is clear) code segment. (See Section 3.4.3.1., "Code- and Data-Segment Descriptor Types", for more information about this flag.)

The rules that the processor uses to check the CPL, RPL, and DPL depends on the setting of the C flag, as described in the following sections.

4.8.1.1. ACCESSING NONCONFORMING CODE SEGMENTS

When accessing nonconforming code segments, the CPL of the calling procedure must be equal to the DPL of the destination code segment; otherwise, the processor generates a general-protection exception (#GP).

For example, in Figure 4-6, code segment C is a nonconforming code segment. Therefore, a procedure in code segment A can call a procedure in code segment C (using segment selector C1), because they are at the same privilege level (the CPL of code segment A is equal to the DPL of code segment C). However, a procedure in code segment B cannot call a procedure in code segment C (using segment selector C2 or C1), because the two code segments are at different privilege levels.

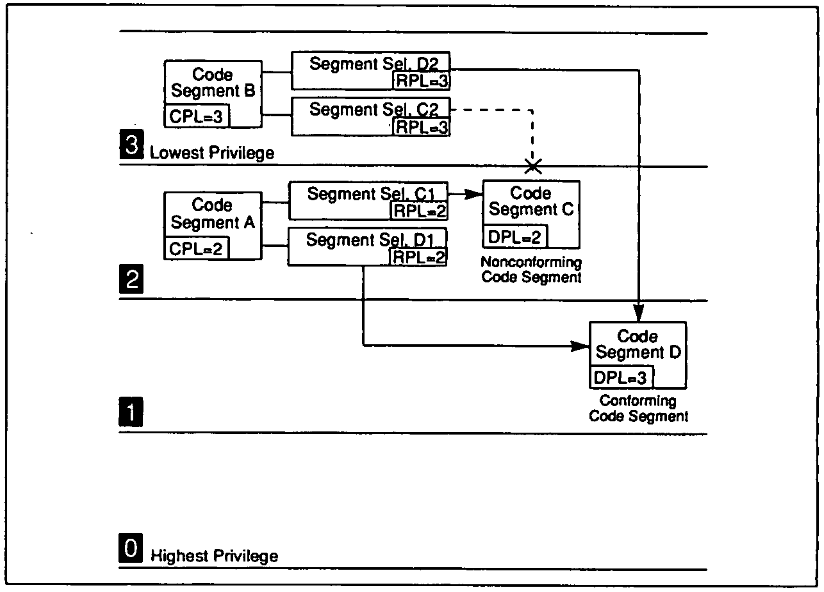


Figure 4-6. Examples of Accessing Conforming and Nonconforming Code Segments From Various Privilege Levels

The RPL of the segment selector that points to a nonconforming code segment has a limited effect on the privilege check. The RPL must be numerically less than or equal to the CPL of the calling procedure for a successful control transfer to occur. So, in the example in Figure 4-6, the RPLs of segment selectors C1 and C2 could legally be set to 0, 1, or 2, but not to 3.

When the segment selector of a nonconforming code segment is loaded into the CS register, the privilege level field is not changed; that is, it remains at the CPL (which is the privilege level of the calling procedure). This is true, even if the RPL of the segment selector is different from the CPL.

4.8.1.2. ACCESSING CONFORMING CODE SEGMENTS

When accessing conforming code segments, the CPL of the calling procedure may be numerically equal to or greater than (less privileged) the DPL of the destination code segment; the processor generates a general-protection exception (#GP) only if the CPL is less than the DPL. (The segment selector RPL for the destination code segment is not checked if the segment is a conforming code segment.)

In the example in Figure 4-6, code segment D is a conforming code segment. Therefore, calling procedures in both code segment A and B can access code segment D (using either segment selector D1 or D2, respectively), because they both have CPLs that are greater than or equal to the DPL of the conforming code segment. **For conforming code segments, the DPL represents the numerically lowest privilege level that a calling procedure may be at to successfully make a call to the code segment.**

(Note that segments selectors D1 and D2 are identical except for their respective RPLs. But since RPLs are not checked when accessing conforming code segments, the two segment selectors are essentially interchangeable.)

When program control is transferred to a conforming code segment, the CPL does not change, even if the DPL of the destination code segment is less than the CPL. This situation is the only one where the CPL may be different from the DPL of the current code segment. Also, since the CPL does not change, no stack switch occurs.

Conforming segments are used for code modules such as math libraries and exception handlers, which support applications but do not require access to protected system facilities. These modules are part of the operating system or executive software, but they can be executed at numerically higher privilege levels (less privileged levels). Keeping the CPL at the level of a calling code segment when switching to a conforming code segment prevents an application program from accessing nonconforming code segments while at the privilege level (DPL) of a conforming code segment and thus prevents it from accessing more privileged data.

Most code segments are nonconforming. For these segments, program control can be transferred only to code segments at the same level of privilege, unless the transfer is carried out through a call gate, as described in the following sections.

4.8.2. Gate Descriptors

To provide controlled access to code segments with different privilege levels, the processor provides special set of descriptors called gate descriptors. There are four kinds of gate descriptors:

- Call gates
- Trap gates
- Interrupt gates
- Task gates

Task gates are used for task switching and are discussed in Chapter 6, *Task Management*. Trap and interrupt gates are special kinds of call gates used for calling exception and interrupt handlers. The are described in Chapter 5, *Interrupt and Exception Handling*. This chapter is concerned only with call gates.

4.8.3. Call Gates

Call gates facilitate controlled transfers of program control between different privilege levels. They are typically used only in operating systems or executives that use the privilege-level protection mechanism. Call gates are also useful for transferring program control between 16-bit and 32-bit code segments, as described in Section 16.4., "Transferring Control Among Mixed-Size Code Segments".

Figure 4-7 shows the format of a call-gate descriptor. A call-gate descriptor may reside in the GDT or in an LDT, but not in the interrupt descriptor table (IDT). It performs six functions:

- It specifies the code segment to be accessed.
- It defines an entry point for a procedure in the specified code segment.
- It specifies the privilege level required for a caller trying to access the procedure.

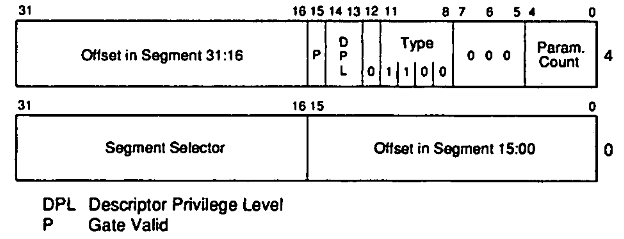


Figure 4-7. Call-Gate Descriptor

protected data structure. This ability to lower the RPL of a segment selector breaches the processor's protection mechanism.

Because a called procedure cannot rely on the calling procedure to set the RPL correctly, operating-system procedures (executing at numerically lower privilege-levels) that receive segment selectors from numerically higher privilege-level procedures need to test the RPL of the segment selector to determine if it is at the appropriate level. The ARPL (adjust requested privilege level) instruction is provided for this purpose. This instruction adjusts the RPL of one segment selector to match that of another segment selector.

The example in Figure 4-12 demonstrates how the ARPL instruction is intended to be used. When the operating-system receives segment selector D2 from the application program, it uses the ARPL instruction to compare the RPL of the segment selector with the privilege level of the application program (represented by the code-segment selector pushed onto the stack). If the RPL is less than application program's privilege level, the ARPL instruction changes the RPL of the segment selector to match the privilege level of the application program (segment selector D1). Using this instruction thus prevents a procedure running at a numerically higher privilege level from accessing numerically lower privilege-level (more privileged) segments by lowering the RPL of a segment selector.

Note that the privilege level of the application program can be determined by reading the RPL field of the segment selector for the application-program's code segment. This segment selector is stored on the stack as part of the call to the operating system. The operating system can copy the segment selector from the stack into a register for use as an operand for the ARPL instruction.

4.10.5. Checking Alignment

When the CPL is 3, alignment of memory references can be checked by setting the AM flag in the CR0 register and the AC flag in the EFLAGS register. Unaligned memory references generate alignment exceptions (#AC). The processor does not generate alignment exceptions when operating at privilege level 0, 1, or 2. See Table 5-6 for a description of the alignment requirements when alignment checking is enabled.

4.11. PAGE-LEVEL PROTECTION

Page-level protection can be used alone or applied to segments. When page-level protection is used with the flat memory model, it allows supervisor code and data (the operating system or executive) to be protected from user code and data (application programs). It also allows pages containing code to be write protected. When the segment- and page-level protection are combined, page-level read/write protection allows more protection granularity within segments.

With page-level protection (as with segment-level protection) each memory reference is checked to verify that protection checks are satisfied. All checks are made before the memory cycle is started, and any violation prevents the cycle from starting and results in a page-fault exception being generated. Because checks are performed in parallel with address translation, there is no performance penalty.

The processor performs two page-level protection checks:

- Restriction of addressable domain (supervisor and user modes).
- Page type (read only or read/write).

Violations of either of these checks results in a page-fault exception being generated. See Chapter 5, "Interrupt 14—Page-Fault Exception (#PF)", for an explanation of the page-fault exception mechanism. This chapter describes the protection violations which lead to page-fault exceptions.

4.11.1. Page-Protection Flags

Protection information for pages is contained in two flags in a page-directory or page-table entry (see Figure 3-14): the read/write flag (bit 1) and the user/supervisor flag (bit 2). The protection checks are applied to both first- and second-level page tables (that is, page directories and page tables).

4.11.2. Restricting Addressable Domain

The page-level protection mechanism allows restricting access to pages based on two privilege levels:

- Supervisor mode (U/S flag is 0)—(Most privileged) For the operating system or executive, other system software (such as device drivers), and protected system data (such as page tables).
- User mode (U/S flag is 1)—(Least privileged) For application code and data.

The segment privilege levels map to the page privilege levels as follows. If the processor is currently operating at a CPL of 0, 1, or 2, it is in supervisor mode; if it is operating at a CPL of 3, it is in user mode. When the processor is in supervisor mode, it can access all pages; when in user mode, it can access only user-level pages. (Note that the WP flag in control register CR0 modifies the supervisor permissions, as described in Section 4.11.3., "Page Type".)

Note that to use the page-level protection mechanism, code and data segments must be set up for at least two segment-based privilege levels: level 0 for supervisor code and data segments and level 3 for user code and data segments. (In this model, the stacks are placed in the data segments.) To minimize the use of segments, a flat memory model can be used (see Section 3.2.1., "Basic Flat Model"). Here, the user and supervisor code and data segments all begin at address zero in the linear address space and overlay each other. With this arrangement, operating-system code (running at the supervisor level) and application code (running at the user level) can execute as if there are no segments. Protection between operating-system and application code and data is provided by the processor's page-level protection mechanism.

4.11.3. Page Type

The page-level protection mechanism recognizes two page types:

- Read-only access (R/W flag is 0).
- Read/write access (R/W flag is 1).

When the processor is in supervisor mode and the WP flag in register CR0 is clear (its state following reset initialization), all pages are both readable and writable (write-protection is ignored). When the processor is in user mode, it can write only to user-mode pages that are read/write accessible. User-mode pages which are read/write or read-only are readable; supervisor-mode pages are neither readable nor writable from user mode. A page-fault exception is generated on any attempt to violate the protection rules.

The P6 family, Pentium, and Intel486 processors allow user-mode pages to be write-protected against supervisor-mode access. Setting the WP flag in register CR0 to 1 enables supervisor-mode sensitivity to user-mode, write-protected pages. This supervisor write-protect feature is useful for implementing a "copy-on-write" strategy used by some operating systems, such as UNIX*, for task creation (also called forking or spawning). When a new task is created, it is possible to copy the entire address space of the parent task. This gives the child task a complete, duplicate set of the parent's segments and pages. An alternative copy-on-write strategy saves memory space and time by mapping the child's segments and pages to the same segments and pages used by the parent task. A private copy of a page gets created only when one of the tasks writes to the page. By using the WP flag and marking the shared pages as read-only, the supervisor can detect an attempt to write to a user-level page, and can copy the page at that time.

4.11.4. Combining Protection of Both Levels of Page Tables

For any one page, the protection attributes of its page-directory entry (first-level page table) may differ from those of its page-table entry (second-level page table). The processor checks the protection for a page in both its page-directory and the page-table entries. Table 4-2 shows the protection provided by the possible combinations of protection attributes when the WP flag is clear.

4.11.5. Overrides to Page Protection

The following types of memory accesses are checked as if they are privilege-level 0 accesses, regardless of the CPL at which the processor is currently operating:

- Access to segment descriptors in the GDT, LDT, or IDT.
- Access to an inner-privilege-level stack during an inter-privilege-level call or a call to in exception or interrupt handler, when a change of privilege level occurs.

4.12. COMBINING PAGE AND SEGMENT PROTECTION

When paging is enabled, the processor evaluates segment protection first, then evaluates page protection. If the processor detects a protection violation at either the segment level or the page level, the memory access is not carried out and an exception is generated. If an exception is generated by segmentation, no paging exception is generated.

Page-level protections cannot be used to override segment-level protection. For example, a code segment is by definition not writable. If a code segment is paged, setting the R/W flag for the pages to read-write does not make the pages writable. Attempts to write into the pages will be blocked by segment-level protection checks.

Page-level protection can be used to enhance segment-level protection. For example, if a large read-write data segment is paged, the page-protection mechanism can be used to write-protect individual pages.

Table 4-2. Combined Page-Directory and Page-Table Protection

Page-Directory Entry		Page-Table Entry		Combined Effect	
Privilege	Access Type	Privilege	Access Type	Privilege	Access Type
User	Read-Only	User	Read-Only	User	Read-Only
User	Read-Only	User	Read-Write	User	Read-Only
User	Read-Write	User	Read-Only	User	Read-Only
User	Read-Write	User	Read-Write	User	Read/Write
User	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Only	User	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write

NOTE:

* If the WP flag of CR0 is set, the access type is determined by the R/W flags of the page-directory and page-table entries.

G

7.4.4. Valid Interrupts

The local and I/O APICs support 240 distinct vectors in the range of 16 to 255. Interrupt priority is implied by its vector, according to the following relationship:

priority = vector / 16

One is the lowest priority and 15 is the highest. Vectors 16 through 31 are reserved for exclusive use by the processor. The remaining vectors are for general use. The processor's local APIC includes an in-service entry and a holding entry for each priority level. To avoid losing interrupts, software should allocate no more than 2 interrupt vectors per priority.

7.4.5. Interrupt Sources

The local APIC can receive interrupts from the following sources:

- Interrupt pins on the processor chip, driven by locally connected I/O devices.
- A bus message from the I/O APIC, originated by an I/O device connected to the I/O APIC.
- A bus message from another processor's local APIC, originated as an interprocessor interrupt.
- The local APIC's programmable timer or the error register, through the self-interrupt generating mechanism.
- Software, through the self-interrupt generating mechanism.
- (P6 family processors.) The performance-monitoring counters.

The local APIC services the I/O APIC and interprocessor interrupts according to the information included in the bus message (such as vector, trigger type, interrupt destination, etc.). Interpretation of the processor's interrupt pins and the timer-generated interrupts is programmable, by means of the local vector table (LVT). To generate an interprocessor interrupt, the source processor programs its interrupt command register (ICR). The programming of the ICR causes generation of a corresponding interrupt bus message. See Section 7.4.11., "Local Vector Table", and Section 7.4.12., "Interprocessor and Self-Interrupts", for detailed information on programming the LVT and ICR, respectively.

7.4.6. Bus Arbitration Overview

Being connected on a common bus (the APIC bus), the local and I/O APICs have to arbitrate for permission to send a message on the APIC bus. Logically, the APIC bus is a wired-OR connection, enabling more than one local APIC to send messages simultaneously. Each APIC issues its arbitration priority at the beginning of each message, and one winner is collectively selected following an arbitration round. At any given time, a local APIC's the arbitration priority is a unique value from 0 to 15. The arbitration priority of each local APIC is dynamically modified after each successfully transmitted message to preserve fairness. See Section 7.4.16., "APIC Bus Arbitration Mechanism and Protocol", for a detailed discussion of bus arbitration.

G

Changed: 8/30/95 Validation TechnologyPrinted: 8/30/95 Page 1/3tel Rin

DRAFT

Instruction Set Architecture Testing

Reed K. Christensenm/s JF1-19, (503) 264-4619 rkc@ichips.intel.com

1. Introduction The main objective of the this project is to advance the technology of architectural-level testing of any **Instruction** SetArchitecture (ISA), so that it is done in a formalized and systematic method. A software tool will be produced that will incorporate the validation technologies developed by the project. The customers for this tool are validators involved in specif ication-based validation. The specification in this case is theISA of the processor. In the past, creating test cases for an ISA has been an informal process of a validator reading the specification and then deciding ad hoc which conditions and combinations should be tested. Since these test cases were not sys-tematically developed, it is no surprise that a suite of this type is characterized as "incomplete" by the validators who use it.

A slight diversion might be worthwhile at this point to def ine what architectural-level testing is, and how one would goabout doing a "complete" job of it. Architectural-level testing consists of the following:

1. setting **architecturally** visible state (registers, memory, flags) to known values 2. executing a single **instruction** from the ISA 3. checking **architecturally** visible state for changes (as **defined** by the behavioral description of the **instruction** justexecuted)

Obviously, this type of testing is only a small part of validating a silicon implementation of an ISA since it simplifies awaythe corner cases introduced by pipelining, caching, out-of-order execution, and a myriad of other features of a real silicon implementation. Good architectural-level testing can be thought of a necessary, but not sufficient condition for good overallvalidation of the processor implementation. is perhaps, the best place to begin the validation process (the top of the validation food-chain), since it is the highest abstract view of the required processor behavior. How does one do "complete" architectural-level testing? The model for architectural-level testing may seem simple, buteven this level of testing is a lar ge enough problem that complete testing does not imply exhaustive testing. For instance, even testing a simple register-to-register MOV **instruction** becomes a very large number of test cases is one were to exhaus-tively check that every number can be successfully moved from every register to every register . Typically data values are formed into equivalence classes, so that testing of one value from the class is considered the same as testing all values ofthe class1. Boundary values from the class are also usually tested.

With data values grouped into equivalence classes, complete architectural-level testing becomes the task of exploring thebehavioral paths in each **instruction**. By traversing a path through an **instruction**, a list of state can be built up which would cause the path to be hit when the **instruction** is executed. Also a list of state which should be altered by the **instruction** canbe collected while traversing the path. The number of paths to be traversed may be too large to be exhaustively explored in the time available. Some means must be provided to prune the path choices and to reduce combinations or cross-productsof things to try.

1. Random testing of values from an equivalence class is often used to test the validator's assumption of equivalence.

Changed: 8/30/95 Validation TechnologyPrinted: 8/30/95 Page 2/3tel Rin

DRAFT 2. DaVinci Use A behavioral description of an **instruction** from the ISA can be written in a formal language, and then represented as agraph. The graph representation is a useful way of visualizing the behavioral paths through the **instruction**. Here is the description of a MOV **instruction**:

(see mov.alg file) The daVinci representation provides a good visualization of the behavioral paths through this **instruction**:

(see mov.daVinci) The software tools provided by this project use daV inci for much more than a static visualization of an **instruction** algo-rithm. The **instruction** graph is loaded by the dgm tool where it can be manipulated by primitive graph routines. These routines are available to the user to be used in a full (T cl) scripting environment. As the graph in memory is modif ied by thevalidator, the corresponding visualization graph provided by daVinci is updated though the daVinci application interface.

The validator is also able to modify the graph in dgm memory by using the daVinci graphical interface. Node(s) can beselected in the daVinci window and a message will be sent from daVinci identifying them. Buttons are provided in a separate Xwindow which will cause the daVinci message to be interpreted to mean dif ferent actions, such a deleting a node,expanding a sub-graph in place of the node, etc. These cause modif ications to the graph in dgm memory, which is, of course, reflected back to daVinci for display.

(see mov_big.daVinci for a modified version of the original mov.daVinci graph)

jmp.graf

Figure 1: iATS Functional Model TclLibrary

TkLibrary ExpectLibrary

daVinci File Window Help

Tcl Interpreter

tcl>

GraphRoutines

g1( )g2( )

Cahners
MICROPROCESSOR REPORT
www.MPRonline.com
THE INSIDER'S GUIDE TO MICROPROCESSOR HARDWARE

Guest Viewpoint: Is Out-of-Order Out of Date?

IA-64's Parallel Architecture Will Improve Processor Performance

By William S. Worley Jr., HP Labs, and Jerry Huck, IA-64 Architecture Lab {2/7/00-02}

Microprocessors are on a relentless path to higher performance. Every innovation in computing—data mining, Java programming, distributed computing on the Internet, multimedia data streams, and so on—invariably requires greater computing power. Even traditional database processing and technical computing have increasing problem sizes that drive demand for higher-performance microprocessors.

To meet these and other future requirements, new approaches to exploit improvements in IC processes are needed. Today, nearly all microprocessors exploit parallelism to accomplish more work in less time. We believe that parallelism can best be exploited with a computer architecture that is designed from the ground up to support instruction-level parallelism (ILP). We have termed this style of architecture EPIC, for explicitly parallel instruction-set computing. IA-64, developed jointly by HP and Intel, is such an architecture (see *MPR 5/31/99-01*, "IA-64: A Parallel Instruction Set").

IA-64 enables the compiler to express more parallelism to the machine than is possible with existing RISC or CISC architectures. As a result, IA-64 significantly reduces the hardware cost of detecting and scheduling the parallelism among instructions. The ability to specify this parallelism directly is one of IA-64's primary advantages.

Through the 1980s, RISC and CISC architectures were not designed primarily for high ILP. Instead, they were designed to make the best use of the technology that was available at the time. The RS/6000 and Alpha architectures adopted similar computing resources and instruction set formats. The inability of RISC and CISC architectures to express parallelism directly can be overcome to some degree by adopting complex, nonarchitectural approaches, principally out-of-order (OOO) dynamic superscalar hardware. Although preserving customer investments and supporting an installed system base are good business reasons for developing such processors for legacy architectures, the advent of IA-64 eliminates the performance of OOO hardware as a compelling technical reason to do so.

The growing market requirement for a higher-performance 64-bit architecture and absence of existing Intel 64-bit binaries gave HP and Intel an opportunity to create something new. The companies took advantage of this opportunity—along with lessons learned from the past 15–20 years of computing evolution—to create a new architecture with performance characteristics superior to those of existing RISC and CISC architectures.

Not Just for ILP

A microprocessor that minimizes computation cycles makes a better building block for high-performance systems. Such a microprocessor can, for example, be replicated to build multiprocessor systems. Fast CPUs in an MP configuration reduce queuing and contention, and they provide greater overall throughput than a larger

G

number of slower processors, a fact that we have seen many times in OLTP benchmarks in which small numbers of PA-RISC processors outperformed larger numbers of slower processors.

The view that parallelism is achieved most effectively through higher levels of multiprogramming is unproved, and, to the extent that it may be true, does not provide the complete picture. It fails to appreciate the fact that parallelism must be improved at all levels of a system. Providing parallelism solely through hardware-based multithreading, simultaneous multithreading (SMT), or chip-level multiprocessing (CMP) cannot compensate for the lack of parallelism in the basic processing element. This is obvious for single-threaded code, but it is true even for some multithreaded code, such as the encryption codes mentioned later.

SMT and CMP apply equally to RISC, CISC, and EPIC microprocessors. The first paper design of an EPIC machine at HP labs in 1991 envisioned integrated hardware multithreading as an orthogonal complement to EPIC architecture capabilities. But SMT has its downsides. The nonlinear nature of caches can be a problem for some workloads. Instead of one thread thrashing the cache on one processor, one thread on an SMT processor can thrash the cache for all threads on the processor. Finding the best design for multiple working sets to share a single cache is a research problem that has generated several papers but, as yet, no clear solutions.

Effective utilization of the hardware resources of a modern microprocessor is difficult. Historically, we have found that doubling the number of function units of a RISC processor has resulted in less-than-linear scaling. IA-64 was specifically designed to utilize additional function units effectively. Defenders of superscalar RISC architectures argue that out-of-order processing is the best means to achieve high function-unit utilization.

Building an out-of-order processor, however, is complex and difficult. The original PA-8000, for example, used as many transistors in its reorder buffer as were used in the entire previous-generation PA-7200 chip. Most current-generation OOO engines are four-issue implementations, and our studies indicate that the complexity of these machines will scale quadratically for 1.5× or 2× increases in issue width. In contrast, the first member of the IA-64 family—Itanium (nèe Merced)—is already a six-issue machine.

Architecture vs. Implementation

Architectural influences are determinative for many parts of an implementation, but not for all. The speed of an ALU, for example, is primarily a function of IC process and word width. The repertoire of operations that the ALU can perform, however, is a second-order issue. In addition, the data-cache hierarchy and the memory system of RISC and EPIC processors are largely independent of the architecture. Cache and memory-system interfaces must meet the demands of the processor—be it RISC, CISC, or EPIC. Criticizing the IA-64 architecture on the basis of an initial memory-system design, which was chosen to balance cost and performance, is a bit unfair.

Some assert that memory systems can be more fully utilized by OOO RISC designs. Our analysis of contention, cache behavior, buffer queuing, processor affinity, memory interleaving, and other factors indicates that one can find better approaches to use available memory technology if one is willing to accept additional cost.

The IC process, the number of registers, the number of register ports, the bypass network, and the number of cache ports are the principal factors in determining the cycle time of an IA-64 processor. Any RISC or CISC design faces similar challenges. The critical path in many modern microprocessors, IA-64 processors included, is found in the function units and their bypass networks. But IA-64 processors distinguish themselves by higher utilization of this fundamental structure. As with all designs, a balance was sought among the clock rate, pipeline depth, and execution width.

The first IA-64 processors will be used in high-end servers and workstations. Over time, designs will broaden out to span a wide range of markets. An HP Labs study for high-performance embedded controllers has confirmed that EPIC-like machines are exceptionally effective.

IA-64's Parallelism Features

Several IA-64 capabilities express and enhance parallel execution. The first is predication, which reduces the number of encountered branches, mispredicted branches, and other obstacles to finding parallelism. Our studies conclude that the gain from branch reduction more than compensates for any extra instructions that might be executed (Note: Do not think of a single if clause, think of merging three to five different basic blocks into a single, branchless, critical-path-limited sequence of code.). A mispredicted branch disrupts the pipeline. The lost opportunity can be measured as the width of the machine times the length in cycles of the mispredict penalty. For newer RISC processors, this is often more than 20 instruction slots.

IA-64 specifies a large register set (128 GP registers and 128 FP registers), a feature that allows algorithm design to be fundamentally changed. Matrix operations, finite-element analysis, and many other technical algorithms can be restructured to take better advantage of the large register space. OOO superscalar advocates argue that their internal register-renaming hardware can bring to bear just as many register resources as IA-64. Rename registers, however, are not as effective as real registers for either the programmer or the hardware.

Having only 32 architecturally visible registers, as most RISC architectures do, requires that a programmer structure his or her code in such a manner that, at any point in time, the registers containing program state do not exceed 32. For problems such as 1,024-bit RSA encryption, one can effectively use many more than 32 general registers and 32

G

floating-point registers. Holding just two 512-bit operands and one 1,024-bit intermediate result fills 32 64-bit registers. For a RISC or CISC processor, the RSA program would require many housekeeping load and store instructions whose only productive functions are to limit the instantaneous general- and floating-point-register state. Although the underlying hardware may have many more internal rename registers, the programmer has no direct means to use these registers to hold program state. Two of the AES-study codes, mentioned later, used over 60 general registers. Having this many architecturally visible resources, and the IA-64 register rotation, enables coding strategies that simply are not possible with an architected set of only 32 registers.

On the hardware side, even though OOO superscalar implementations normally have extra internal registers, during every cycle they must make visible only the 32 registers of program state. Furthermore, in the event of an interruption, the hardware must be prepared to lose (and later perhaps partially reconstruct) all the nonvisible internal state. Thus, as is the case for the programmer and for the executable code, the hardware's use of additional register resources is handicapped by the need to maintain the fiction that the sole register state is that constituted by the 32 registers.

Most significant, IA-64 introduces a collection of features to deal with memory latencies, which continue to increase relative to processor speeds. IA-64's control and data speculation capabilities enable compiler-directed access to variables at points much earlier than they are needed for computation. This capability permits a greater degree of concurrency between executing instructions and memory accesses. OOO engines achieve a similar effect with dynamic hardware, but they are restricted to fixed hardware algorithms for correctly predicting the execution path and for triggering memory fetches. HP's analysis of the PA-8000 shows that the primary benefit of OOO operation in commercial workloads lies in initiating multiple earlier cache misses. IA-64 enables such acceleration on an even broader scale.

Involving the compiler in the process of identifying speculative load candidates opens a bigger window into the program than can practically be achieved by an OOO superscalar processor. Data profiling makes the compiler even more accurate at selecting variables for speculative handling. The IA-64 compiler has heuristics to control the degree of speculation, and the programmer has control over the compiler's heuristics.

Another important feature of IA-64 is its register stack engine (RSE). One might consider this feature a built-in asynchronous hardware thread that runs when there would otherwise be idle memory ports. This feature reduces the cost of procedure calls and returns and increases the utilization of the register file. It is especially valuable for accelerating call-intensive object-oriented code. The reduction in the time to spill and fill the general register file is significant for many applications. On one database benchmark, for example, RISC processors spend about 30% of their memory references for procedure entry/return housekeeping. Most of this overhead is eliminated by IA-64's RSE. The IA-64 architecture has been crafted carefully to make the hardware design of the RSE straightforward. The RSE does not add to the critical path of the machine and is a relatively small part of the Itanium and McKinley designs.

Putting It All Together

All of the IA-64 elements mentioned above combine synergistically to minimize the critical code path through a program. The size of the resulting program binary image may be larger, but IA-64's instruction stream is more linear, i.e., it contains fewer branches. Itanium and McKinley both compensate for this code growth with special mechanisms that efficiently deliver instructions to the processor. These mechanisms eliminate the effects of the increased code size with only modest area and design costs.

In a paper submitted to the NIST AES3 (advanced encryption standard) Conference by HP Labs researchers, the five final AES algorithms were analyzed for both PA-RISC and IA-64. This study shows that IA-64's register file and its wide parallel architecture are very effective. The full issue width of the machine was utilized by some of these codes. Not surprisingly, the two algorithms with the greatest theoretical parallelism—Twofish and Rijndael—showed the greatest function unit utilization. Eight of the 15 IA-64 codes (encryption, decryption, and keying for each of the five finalists) used more than 32 registers. Six of the 15 IA-64 codes had smaller code sizes than PA-RISC, due to the compact modulo-scheduling support in IA-64. In two cases, the code was more than four times smaller. Overall, the IA-64 code size was only 27% larger than PA-RISC, even though no explicit effort had been made to minimize code size on either architecture. (A quick look at the IA-64 code showed that the difference could have been reduced to the 10% range.)

Although these are not typical codes, they illustrate how RISC code compares with IA-64 code using identical goals and algorithms. Equivalent comparisons using compiled code are not yet available, because IA-64 compilers are still maturing. The AES study goes into detail on mapping the final AES algorithms onto PA-RISC and IA-64 machines and architectures. Features like predication and rotation played important parts in reducing IA-64 critical code paths and exposing parallelism. In one simple example, recurrence was trivially handled by referencing back into the rotating-register region—no extra copy or unrolling was needed.

IA-64 Compilers Find Parallelism

IA-64 builds on proven compiler techniques to extract parallelism from applications. Many of these techniques are used by existing compilers to gain the best performance for today's OOO RISC systems. These techniques include data prefetch,

G

branch hints, loop unrolling, and profile-based path structuring, as well as other well-established optimizations.

As an example, OOO RISC compilers generate better code when using profile results. On the PA-8000, reducing the number of taken branches, through profiling, improves branch prediction. This is especially valuable in large commercial codes, where branch prediction tables are not very effective.

Historically, every major improvement in instruction-level parallelism has required the use of new code-generation techniques. Only with such techniques can the compiler realize significant performance gains. The PA-8000 and Alpha 21264, for example, used new binaries to get optimal performance. The initially-hoped-for transparent performance gains from OOO superscalar machines have not materialized. Meticulous code scheduling by the compiler has proved just as essential for OOO engines as it will be for IA-64.

Code profiling is only slightly more important for IA-64 code generation than it is for an OOO processor. Without profiling, an OOO engine can easily get lost down the wrong path, due to branch mispredictions and false dynamic speculation—especially in large-footprint applications. Instructions that will not be executed are cached, and in-flight instructions are canceled.

As a further example, the performance of the specFP95 benchmark is greatly improved by inserting prefetch instructions. The OOO engine is not effective in automatically triggering the proper prefetches. OOO queues are generally not deep enough and do not understand which data will be needed. On the other hand, the compiler is able to analyze the data layout and trigger the best prefetches.

Compiler writers, and those who have hand-coded for OOO machines, talk of the frustration in understanding how to second-guess the limitations of the OOO hardware and work around them to achieve full utilization of the function units. The job usually boils down to trial and error. This process actually occurred for PA-RISC codes during the AES study.

Another significant issue with OOO design is sustaining the most critical memory references in flight. To the OOO engine, everything is equal. The IA-64 compiler, on the other hand, is able to locate the critical path through the code and to ensure that the important long-latency operations are started first. Memory buffer and other limitations will always mandate executing critical path instructions first. As noted in an earlier paragraph, the compiler will issue prefetches to ensure early initiation of the most-likely cache misses.

To complement the compiler's expanded ability to avoid cache misses, hardware resources still can be brought to bear in an IA-64 implementation. As an example, we have developed several strategies to improve the handling of cache misses. An in-order machine has options beyond a simple stall when a cache miss occurs. Running ahead with rollback, predictive address buffers, implicit prefetch, buddy prefetch, and other dynamic approaches can all be effective. Quantitatively, these techniques are far simpler than those used in OOO designs.

Static code generation is just one aspect of producing efficient code. The recent announcement of the Crusoe processor by Transmeta hints at the benefits of dynamic code generation. In many venues, researchers have been examining the significant performance improvements that can be achieved by dynamic measurement and tuning of code.

HP, for example, implemented a simple mechanism that sampled the current instruction during the normal timer interrupt. If the instruction was a kernel branch, the branch hint was rewritten to match the actual program flow. This one dynamic mechanism resulted in a 5% improvement for database applications.

More aggressive approaches are possible. The Itanium processor is able to watch a program's cache misses, and software can apply a simple heuristic to set prefetch hints correctly. This type of explicit approach, driven by runtime information, is extremely accurate and selective. Better performance feedback enables greater tuning accuracy and better utilization of the machine resources. IA-64 has hint fields in most branch and memory-reference instructions. Armed with runtime information for the executing program, hint fields can be rewritten on the fly to match the needs of the current workload.

These approaches allow software to tune code dynamically for performance. With such techniques, and without recompilation, performance can continue to evolve and improve, even after a machine and application has been put into production. IA-64 provides a greater range of tuning options than previous architectures.

Future Directions

The first IA-64 microprocessor, Itanium, exhibits ILP beyond that achievable by any OOO superscalar microprocessor in existence. Extensive study and analysis went into the EPIC architectural ideas, and they have been found to work. Itanium is just the initial implementation of the IA-64 architecture. It delivers the powerful EPIC innovations while providing complete binary compatibility with IA-32 and PA-RISC.

Future designs will be even more powerful. We are now at just the beginning of the IA-64 hardware and software implementation learning curves. As was the case for RISC and CISC architectures, the IA-64 architecture will evolve and become even more powerful. As was the case for RISC and CISC implementations, IA-64 implementations will mature and evolve through successive generations. And, as was the case for RISC and CISC software, the EPIC compilers will become better and better at exploiting the full capabilities of the architecture.

Any initial implementation of a new architecture will be conservative and will concentrate on the most important architectural elements. Itanium and McKinley are not

G

exceptions to these rules. The initial version of the IA-64 architecture by no means encompasses all the innovations and ideas developed by HP and Intel. The initial hardware implementations by no means embody all the techniques and designs envisioned by HP and Intel. Since the early 1990s, HP Labs has been evaluating scalability for high ILP, multiprocessing, hardware multithreading, and high-bandwidth memory systems. HP has also considered many static and dynamic compilation and simulation techniques.

IA-64 Will Deliver More ILP and Performance

IA-64 will deliver on its promise of expressing, enhancing, and exploiting instruction-level parallelism to improve performance. The IA-64 architecture will not remain static or fixed, and successive generations of processors will each introduce innovations. By the time we have third- and fourth-generation chips, we are confident that the present architectural controversy will have passed, and EPIC will have proved its superiority.

Bill Worley is a principal architect on two HP architectures: PA-RISC and PA Wide-Word, the later of which became the basis for HP's collaboration with Intel on IA-64. In 1995, Bill was named a distinguished contributor, the highest technical position at HP. Bill can be contacted at worley@hpl.hp.com.

Jerry Huck manages processor architecture in HP's computer products organization. These days his time is split between working with Intel on the IA-64 architecture and managing the team responsible for processor simulator and platform architecture definition. Jerry's team was also responsible for the evolution of the PA-RISC architecture. Jerry can be reached at jerry_huck@hp.com. ◇

G

BRIDGING THE PROCESSOR-MEMORY GAP

David Patterson and Katherine Yelick
Computer Science Division
EECS Department
University of California, Berkeley
Berkeley, CA 94720-1776

Final Report 1999-00 for MICRO Project #99-094
Industrial Sponsors: Microsoft, Micron Technology, SGI/CRAY Research,
Seagate Technology, and Texas Instruments

ABSTRACT: Two trends call into question the current practice of microprocessors and DRAMs being fabricated as different chips on different fab lines: 1) the gap between the speed of processor and the speed of DRAM is growing at 50% per year; and 2) the size and organization of memory on a single DRAM chip is becoming awkward to use in a system, yet size is growing at 60% per year. Intelligent RAM, or IRAM, merges processing and memory into a single chip to lower memory latency and increase memory bandwidth as well as to select the best memory size and organization for an application. In addition, IRAM promises savings in power and board area.

1. INTRODUCTION

The IRAM project is architecting, fabricating and evaluating a single chip supercomputer that combines a configurable processor and high capacity DRAM to deliver vector supercomputer-style sustained floating point and memory performance, at vastly reduced power. This chip will be called "IRAM", standing for Intelligent RAM, since most of transistors on this merged chip will be devoted to memory. The goal is to demonstrate that a single chip with a simple processor and a very high bandwidth local memory can be faster on memory-intensive problems as well as be a much better match to real-time applications. Given that conventional machines will have separate chips for the processor, external cache, main memory, and networking, an IRAM would also be smaller, use less power, and be less expensive. The design targets the gigabit generation of DRAM, which offers 128 megabytes per chip.

A second objective of this project is to design and prototype a multi-chip system in which processing is added to various levels of the storage system, called Intelligent Storage (ISTORE). Such a system is suited to I/O intensive applications such as decision support databases, when the IRAM chip is placed inside each disk to eliminate the I/O bottleneck of centralized servers. It is also designed for large memory-intensive computations that are too large for a single chip.

Project web page: http://iram.cs.berkeley.edu/

2. PROGRESS

The IRAM group made progress on several fronts over the past year.

3. CHIP DEVELOPMENT

3.1 VIRAM Testchip

The test chip was submitted to our industrial partner, LG Semicon, in November 1998. The chips were originally scheduled for fabrication in March, but a strike at LG Semicon has delayed the chips until June. Problems with the DRAM fab continued, and Hyundai purchased the DRAM portion of LG. Although we finally received parts, they were so delayed and the DRAM is known to be flawed so as to make the value of the test chip questionable. We consider this phase completed, and do not expect to continue any significant effort on this part of the project.

3.2 Serial I/O Test Chip

The I/O testchip was successfully completed in 1998, as described in the last report. The power and area results are considerably less than previously published 1 Gbit/sec serial links (400 mW and 4.4 mm2 in a 0.5 um process.) We consider this phase completed, and do not expect to continue any significant effort on this part of the project.

3.3 VIRAM-1

We continued to investigate relationships with TSMC and IBM as our industrial partner for the final VIRAM-1 chip. We went forward on a partnership with IBM, which looked very promising from both a technical perspective (better die size than TSMC) and because of a strong commitment on their end to the IRAM project. We have received some of the technical information from IBM and have signed a temporary nondisclosure agreement to get additional information such as Spice models for the DRAM. We also worked in a parallel interface for VIRAM-1 to simplify interfacing to other devices.

On May 1, 1999 we completed a preliminary management agreement with IBM and obtained data about the IBM DRAM macro and other chip level specifications that are needed for the final VIRAM-1 design.

The chip will contain DRAM memory, a scalar processing core, a floating-point co-processor, a vector unit with 6 64-bit pipes (which can also be subdivided into 32 or 16 bit lanes) and a network interface (referred to as parallel I/O lines in previous reports). The high level design of each of these

G

components was completed prior to this year, with the exception of the network interface and a DMA engine; the network interface was completed and the DMA design is in progress. We completed the Verilog (RTL) models for most of the control for the memory pipeline and the integer and floating-point units. Some details that involve the TLB and floating-point interfaces remain to be tested. We have also completed the circuit design and layout of the integer multiplier, which is based on a modular design involving a 16-way repetition of a single circuit block.

We have also developed a verification framework for testing the lower level designs and, eventually the chip. The ISA simulator is used as the "golden model" against which the other designs are tested. The verification framework allows for three basic kinds of tests. 1) Self checking provides the output (in terms of register/memory values) which is compared against the values produced by a simulator. 2) Trace comparison is used to compare the traces of any two simulators. These are based on traces of the architecturally visible state, i.e., instructions, PC, registers, and values read/written to particular memory addresses. 3) Directed tests are used for testing state that is not visible from the architectural level and may therefore not be implemented in a higher-level simulator like the ISA simulator. An example is monitoring the cache replacement policy by inspecting the contents of the cache directly.

3.4 Progress on VIRAM Architecture

One of the major agreements was to find a partner to supply the MIPS scalar processor. That company was Sandcraft, and they will supply their next generation embedded MIPS processor. This is a full MIPS IV CPU, including floating point, TLB, and caches. Sandcraft has given permission for us to use their floating-point unit and TLB in our vector coprocessor, considerably reducing our tasks.

We wrote a paper related to the motivation for IRAM, making a case for architecture research personal mobile computing, where portable devices are used for visual computing and personal communications tasks [6]. The requirements placed on the processor in this environment are energy efficiency, high performance for multimedia and DSP functions, and area efficient, scalable designs. We examined the architectures that were recently proposed for billion transistor microprocessors, and although they are very promising for the stationary desktop and server workloads, we discover that most of them are unable to meet the challenges of the new environment and provide the necessary enhancements for multimedia applications running on portable devices.

3.5 VIRAM Instruction Set Architecture

Based on simulation results from last Fall, we added new instructions to the IRAM ISA to improve the speed of reduction operations. In the original ISA, reductions were done with an "extract" instruction that took a set of values in one vector register and moved them to another vector register. The instruction was fairly general and could involve inter-lane communication, which made it difficult to determine which instance of the instructions could be chained. (In the VIRAM-1 implementation, for example, there are 4 lanes, and the extract instruction required a general crossbar style communication between them.) As a result, the instruction was slow. In the new ISA, there is an extract instruction that moves the upper half of one vector register to another and is only defined for power-of-2 vector lengths. As a result, there is no inter-lane communication as long as the vector length is greater than the number of lanes, which makes the chaining decision easy to implement. In a reduction operation on an implementation with 4 lanes, only the last two extract will involve inter-lane communication. These new instructions were added to the ISA simulator, and after the performance model is modified to match expected performance, we will rerun some of the reduction-intensive benchmarks.

A second issue on which progress continue is the exception handling model. Because we are using a floating-point core from Sandcraft in both the RISC scalar processor and the vector co-processor, some of the exception handling is out of our control. We have determined that infinities and NANs will propagate, which is a good match to the vector processing model, but some other issues such as denormalized numbers have yet to be resolved.

3.6 Functional ISA Simulator

The functional ISA simulator was updated to reflect some new instructions, including the extract described above. In addition, we built a debugging version of the simulator to aid in debugging assembly language programs. The debugger allows programmers to set breakpoints and examine the architectural state, such as register values. This tool has proven invaluable in developing benchmarks.

3.7 Performance Simulator

Graduate students in the graduate architecture course used the performance simulator during Fall 1998. Based in part on feedback from those users, a significant redesign of the system was done to make it more extensible and allow for further investigations of the implementation parameters. Several new internal releases were done. In addition, some output has been added to display the internal pipeline states, which is critical in helping find performance bugs in assembly language code. By visualizing the pipeline, one is able to determine the cause of stalls, such as memory bank conflicts, structural hazards, TLB misses, and gives the programmer some idea of how the instructions might be rearranged to improve performance.

4. SOFTWARE TOOLS

4.1 V-IRAM Compiler

A former member of the SGI/Cray compiler team, Dave Judd, recently joined the IRAM project to work on a port of the Cray compiler to the VIRAM architecture. He began studying the current code generator (which was been replaced) and learning about related compiler efforts within SGI/Cray. Unlike the ISUIF compiler, there was no MIPs backend for the scalar code, so the first milestone was a complete MIPs backend. The next step will be to add vector instructions to the backend; the compiler already performs automatic vectorization, including outer loop vectorization which is important for many applications, so the main problems are to modify code generation and change instruction scheduling to match the performance characteristics of VIRAM-1.

4.2 Benchmarks and Applications

We continued working on the two application efforts in speech and video processing. For the speech application, we are working with the current Cray vectorizing compiler on an existing Cray machine. Although we cannot get detailed performance information about VIRAM, this development effort will give us basic information such as how to annotate loops and how well the overall application vectorizes.

Our second application in video processing uses some of the multimedia features of IRAM, which are unlikely to be supported by the Cray compiler, so some of the kernels are being written directly in VIRAM assembler. We have compiled the modified H.263 and MPEG-2 source code using a MIPS scalar compiler and run both on the IRAM simulator. (H.263 is another standard of video compression used often in video conferencing.) We are in the process of replacing the key computations with hand-vectorized kernels, starting with the Square of Absolute Difference (SAD) used in motion estimation. We implemented SAD in VIRAM assembly language and have started a performance simulation study of different algorithms. We plan to complete this study, taking advantage of the new extract instruction for reductions.

At a higher level, we also explored different motion-estimation algorithms (all of which use SAD) for IRAM. Most of the work was concentrated on three methods: Exhaustive Search, searching through all the macro-blocks; Three-step Search, limiting the search of macro-blocks in a hierarchical fashion; 2D Log Search, the same as three-step search but with somewhat less computation. In addition to motion-estimation, different schemes to speed up the DCT have been investigated. The first one is the Zero Coefficient Prediction Scheme: if the coefficient of a macro blocks are all zeros then there is no need to compute DCT for that block. In addition, different types of fast DCT algorithms were considered. Using rough timing estimates, all of these algorithms have roughly the same performance on VIRAM-1, so more investigation will be needed using the performance simulator.

In addition to these long-term application efforts, we developed some additional benchmarks written directly in VIRAM assembler. The new benchmarks include reduction operations using the new instructions, 3 basic image operation from multimedia (Chroma-Key, image composition, and color conversion), convolutions, FFTs, and matrix-vector multiplication. (The latter three had been written using the Vic tool, but were recoded in the new ISA by hand.) In addition, there are some benchmarks, such as encryption, that are available from ISUIF. So far, these benchmarks have been especially useful at helping to make the simulators most robust and usable.

5. ISTORE

In two years ago we have found another application of IRAM, which has been so interesting that it has taken something of a life of its own. ISTORE is a server architecture with goals quite different from prevailing wisdom:

1) *Maintainability*: ISTORE is intended to have a low cost of ownership, which implies extensive monitoring to discover errors, a physical design that makes repair easy and obvious, the ability to insert faults to test monitoring features, and so on.
2) *Availability*: We believe that a highly available system will also be easier to maintain, as it can decouple the failure of a component from the need for a person to fix the machine immediately. Although general solutions from companies like Tandem work well (mirroring, process pairs, and so on), we are looking for more economical solutions. Our hope is that by tailoring to the software to a single application we can simplify the complexity of the solutions.
3) *Evolutionary Growth*: We want a system that can scale well beyond sizes of today's servers, to thousands or even tens of thousands of disks. As ISTORE leverages cluster technology, we expect to have multiple generations of hardware over time, and hence an emphasis on heterogeneous

systems that evolve over time. Evolutionary growth is in contrast to traditional definition of scalability, which only promises the ability to construct different-sized homogeneous systems.

We abbreviate these three goals as *AME*.

The insight is that IRAM offers a such small, low power computer that it could be included in a canister with a disk at no practical increase in the size of the canister or in the cooling requirements of the canister. Then rather than use the SCSI interface of the disk to connect the disk to the backplane, the serial lines of IRAM would be connected to redundant network interfaces, which would be the connection of the canister to the backplane. The backplane would then be based on single-chip crossbar switches to offer high bandwidth connections between ISTORE nodes.

By having a computer in every I/O building block, we can monitor the behavior of every disk, and provide new features such as fault insertion (to test behavior of operating system software often activated only in failures in the field) and fast failure (to ensure that a suspect device is shut down immediately rather than continue to sporadically make errors.)

5.1 ISTORE Hardware

The IRAM group is moving forward with plans to build a large system for prototyping ISTORE. This will be a 80-node system designed for high reliability as well as performance; each node is physically packaged as a "brick." The node processor is the Intel Mobil Pentium II operating at 366 MHz. Memory is incorporated using small outline DIMMs (SODIMMs) with a target capacity of 128 MB/node. Both of these technologies, originally developed for portable PCs, are well suited to the space and power constraints of the brick.

For inter-node connectivity, we are including four 100 Mb/s Ethernet ports on each brick. Three ports (per brick) will connect to a two-level network built from commercial Ethernet switches. Our design for this network includes 3-way redundancy at the first switch level and 2-way at the second level. A low profile SCSI disk drive (target: 18GB) and SCSI controller chip will be included in each brick. The design will also include the various chips required to make each brick "PC compatible", simplifying the software-porting task.

To perform realtime monitoring of the node processor, we are including a second embedded diagnostic processor with its own communication network. The processor we have chosen is in the Motorola 68K family and includes (on-chip) a protocol controller for the CAN (Controller Area Network) network. This network, originally developed for automobiles, is a good match to this application. The diagnostic processor will monitor environmental conditions (temperature and voltage), communicate with the node processor and also allow us to perform some "intrusion" experiments.

Construction of the ISTORE-1 prototype, targeted to be an 80-node system, continued. Anigma, a small company in southern California, is designing the boards for the nodes. We received the first version of these from an initial "test run" of 10 boards. Anigma found a glitch in the SCSI interface, which they plan to fix over the next month, and in the mean time we will be testing the other parts of the board. In particular, the board contains a diagnostic processor and set of sensors, which were designed at Berkeley. To utilize the diagnostic processor, we are building a simple custom OS, which uses a round-robin scheduler to support multiple kernel threads and processes. Critical error conditions, including upcoming power loss, are handled asynchronously, so a priority scheduler is not necessary (eliminating priority inversion as a source of starvation). Processes will be downloaded at run time, and will communicate through the attached serial and network links. They will also provide failure-resistant intelligence, such as demanding that the OS shut down the brick if multiple neighboring nodes are overheating. The OS will also use a portion of its battery-backed SRAM as a time-stamped log, which will durably record error conditions.

5.2 ISTORE Software

Since our last progress report we have refined our vision of the ISTORE software architecture. The ISTORE software will provide a comprehensive framework for constructing *introspective* applications, i.e., adaptive software that leverages continuous self-monitoring. The ISTORE runtime system will automate the collection of monitoring data from the hardware bricks and will provide applications with customized, application-specific views of that data; it will then allow applications to define triggers over those views that automatically invoke adaptation routines when application-specific conditions are met. For common adaptation goals, the ISTORE system software will go further by providing an extensible mechanism for automatically generating monitoring and adaptation code based on application policy specifications expressed as constraints in declarative, domain-specific languages. Through this mechanism, an application designer will obtain the full benefits of an application-tailored introspective runtime system without having to write large amounts of code and without having to resort to ad-hoc techniques.

As a first step towards understanding how application behavior is affected by low-level system operation and how the system might adapt its low-level behavior to meet application performance and reliability goals, we are instrumenting a single-node operating system to collect detailed realtime system statistics that will then be analyzed off-line using data mining software. This will help to guide us in determining what system statistics should be monitored to drive adaptation, and in understanding how database technology, particularly embedded databases like Berkeley DB, can be used to simplify the organization and processing of the statistics gathered.

We have continued work on performance robustness, to ensure that the performance of the system degrades only gradually as components fail or slow down. For example, as disks become fragmented, the observed performance drops, resulting in a kind of performance heterogeneity in the system. For both performance robustness and for our work on AME characteristics, we have identified several small application projects to be completed in the next two quarters. The first is a web server that uses the Apache software running on a modified version of the Free BSD OS. The second is a mail server; we are currently exploring two different system designs, one involving the Petal software described in a previous report. The third application is database kernels, especially for decision support workloads including data mining.

Unlike the mail and web services, the data mining and related workloads involve a set of closely coordinated processes that will run on each of the ISTORE nodes. For programming these applications, we want a system that helps programmers avoid the kinds of programming errors, such as uninitialized pointer dereferences, memory leaks, and out-of-bounds accesses on buffers that lead to both security and robustness problems. We are therefore looking at a dialect of Java, called Titanium, with extensions for high performance I/O. Previously, Titanium's runtime support for file I/O was limited to the classes present in Java, which have proven too inefficient to meet the demands of I/O-intensive applications. We have developed a new library that adds support for bulk I/O in both synchronous and asynchronous forms. The synchronous model is simpler to use, but the asynchronous model allows for overlapped I/O. These libraries remove much of the overhead associated with the Java I/O libraries, which require the programmer to read arrays a single element at a time, but ensure the same safety guarantees that Java provides. Our performance shows 1-2 order of magnitude improvement over the existing I/O facilities.

Benchmarks have historically played a key role in guiding the progress of computer science systems research and development, but have traditionally neglected the areas of maintainability, availability, and evolutionary growth. These three areas, which we refer to as AME, have recently become critically important in high-end system design. As a first step in addressing this deficiency, we introduced a general methodology for benchmarking the availability of computer systems. Our methodology uses fault injection to provoke situations where availability may be compromised, leverages existing performance benchmarks for workload generation and data collection, and can produce results in both detail-rich graphical presentations or in distilled numerical summaries. We apply the methodology to measure the availability of the software RAID systems shipped with Linux and Windows 2000 Server, and find that the methodology is powerful enough to not only quantify the impact of various failure conditions on the availability of these systems, but also to unearth their design philosophies with respect to transient errors and recovery policy. Finally, we show how these availability benchmarks form a possible foundation for the construction of general maintainability benchmarks. This work is described in a paper to be presented at Usenix 2000 [5].

6. PROJECT MEETINGS

In October we had a meeting with representatives from Atlantic Aerospace and Boeing to discuss the benchmarking effort for the Data-Intensive computing program.

We held a joint project review retreat with the Berkeley Reconfigurable Systems group (BRASS) - January 14-16, 2000. Students working on the project and on research-related class projects presented their working results for our industry affiliates. The representatives of each company were as follows: David Anderson (Seagate Technology), Krste Asanovic (MIT), Bill Athas (USC / ISI), Mike Beunder (Silicon Access Technology), Ray Chen, (Network Appliance), Jack Choquette (Sandcraft), Jason Golbus (Myricom), James Hamilton (Microsoft), Naohiko Irie (Hitachi), David Kiefer (Silicon Graphics), Jongbok Lee (LG Semicon), Mike McGregor (Micron Technology), Todd Merritt (Micron Technology), Steve Scott (Silicon Graphics), Harvey Stiegler (Texas Instruments, Inc.), Jack Veenstra (Sandcraft), Sanjay Vishin (Avaj), Hing Wong (Silicon Access), Mike Ziegler (Hewlett-Packard Company)

7. CONCLUSION

This project made significant progress on all aspects of the project, from chip design through applications software, and continued work on application so important that it has almost taken a life of its own. We hope the project will continue for the next several years.

8. PUBLICATIONS

[1] Aaron Brown, David Oppenheimer, Kimberly Keeton, Randi Thomas, John Kubiatowicz and David A. Patterson, "ISTORE: Introspective Storage for Data-Intensive Network Services." Presented at HotOS-VII, 1999.

[2] R. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. Culler, J. Hellerstein, D. Patterson, K. Yelick, "Cluster I/O with River: Making the Fast Case Common." IOPADS '99.

[3] Randi Thomas and Katherine Yelick, "Efficient FFTs on IRAM." Presented at the 1st Workshop on Media Processors and DSPs.

[4] T. Nguyen and A. Zakhor and K. Yelick, "Performance Analysis of an H.263 Video Encoder on VIRAM." Submitted to ICIP 2000. An earlier version of this appeared as a MS Report by the first author in December 2000.

[5] A. Brown and D. Patterson. "Towards Maintainability, Availability, and Growth Benchmarks: A Case Study of Software RAID Systems," To appear in Usenix 2000.

[6] D. Bonachea. "Asynchronous Bulk File I/O in Titanium, a High Performance SPMD Java Dialect" Submitted to JavaGrande 2000.

[7] Richard M. Fromm, "Vector IRAM Memory Performance for Image Access Patterns." Master's Report, University of California, Berkeley, Computer Science Division, October 1999.

[8] Christoforos Kozyrakis, "Media-Enhanced Vector Architecture for Embedded Memory Systems." Master's Report, University of California, Berkeley, Computer Science Division, July 1999. Appeared as Technical Report UCB//CSD-99-1059.

G

PATENT DOCKET NO. 5231.16-4004C

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicant:	John S. Yates, Jr., et al.		
Serial No.:	09/429,094	Art Unit:	2155
Filed:	October 28, 1999	Examiner:	David Eng
Title:	SIDE TABLES ANNOTATING AN INSTRUCTION STREAM		

INFORMATION DISCLOSURE STATEMENT

COMMISSIONER FOR PATENTS
Washington, D.C. 20231

In accordance with 37 C.F.R. §§1.56, 1.97 and 1.98, Applicant wishes to make of record the enclosed items, as listed on the accompanying Form PTO-1449. Applicant respectfully requests the Examiner to fully consider the items and independently ascertain their teaching before issuance of the next action, and to make them of record in the file. The Examiner is also requested to initial and return a copy of the enclosed Form PTO-1449 to evidence such consideration.

Applicant has listed publication dates on the attached Form PTO-1449 based on information presently available to the undersigned. However, the listed publication dates should not be construed as an admission that the information was actually published on the date indicated. Applicant reserves the right to establish the patentability of the claims over any information provided herewith, and/or to prove that this information may not be prior art, and/or to prove that this information may not be enabling for the teachings purportedly offered. This Information Disclosure Statement should not be construed as a representation that information more material to the examination of this application does not exist.

This is a resubmission of a reference previously brought to the Examiner's attention in IDS's filed November 21, 2000, August 23, 2001 and February 4, 2002. No fee is due for this Information Disclosure Statement because this reference was previously filed in this application with the Information Disclosure Statement of November 21, 2000, for which the proper fee was paid.

The Commissioner is hereby authorized to charge any additional fees that may be required for this Information Disclosure Statement, or credit any overpayment, to Deposit Account 50-0675, Order No. 5231.16-4004C.

Respectfully submitted,

SCHULTE ROTH & ZABEL

Dated: July 8, 2002

By: ______________________
David E. Boundy
Registration No. 36,461

CORRESPONDENCE ADDRESS:

SCHULTE ROTH & ZABEL
919 Third Avenue
New York, New York 10022
(212) 756-2000
(212) 593-5955 Facsimile

I certify that this correspondence, along with any documents referred to therein, is being deposited with the United States Postal Service on July 8, 2002 as First Class Mail in an envelope with sufficient postage addressed to The Commissioner for Patents, Washington D.C. 20231.

____

9

THE 36-V CAR BATTERY ◦ DEALING WITH ENCRYPTION ◦ TAKING ON MICROSOFT

MAY 2000

SPECTRUM

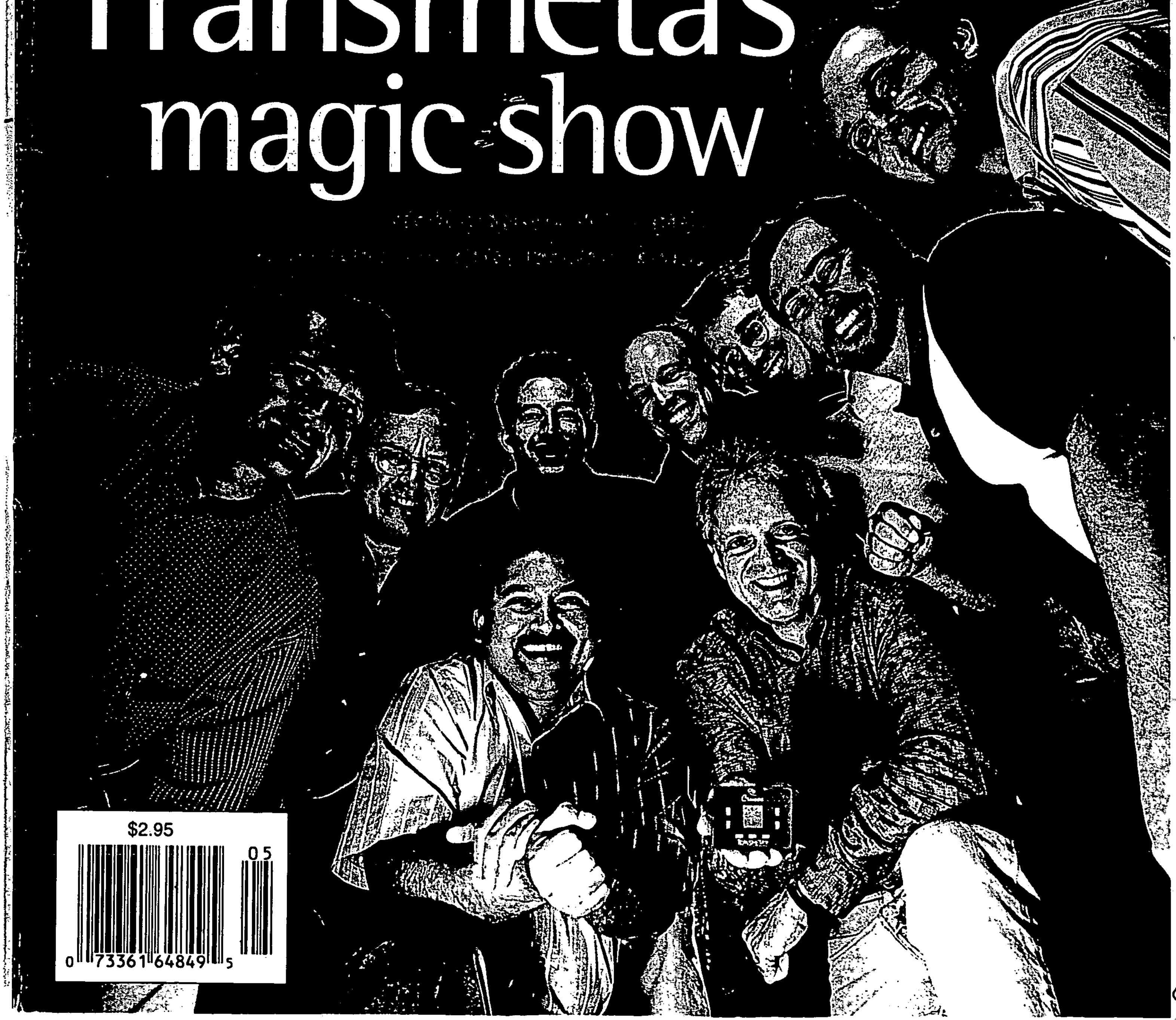
THE INSTITUTE
OF ELECTRICAL
AND ELECTRONICS
ENGINEERS, INC.

http://www.spectrum.ieee.org

Transmeta's magic show

[illegible]

[illegible]



cover story

It took Transmeta engineers $100 million, five years of secret toil, and a little magic to create fast low-power chips that turn into x86s in a microsecond

MAGIC SHOW

LINDA GEPPERT
&
TEKLA S. PERRY
Senior Editors

TRANSMETA CORP.'S CRUSOE CHIPS, due to ship in May or June, look nothing like Intel Corp.'s Pentium processors. In fact, they do not even have a logic gate in common. They are smaller, consume between one-third and one-30th the power (depending on the application), and implement none of the same instructions in hardware.

But the Crusoe microprocessors [Fig. 1] can run the same software that runs on IBM PC-compatible personal computers with Pentium chips—for instance, Microsoft Windows or versions of Unix, along with their software applications.

That's the magic trick. And it took a bunch of engineering magicians—and over $100 million of venture capital—to pull it off.

Transmeta's magic show started more than five years ago. David Ditzel, then the chief technical officer of Sun Microsystems Inc.'s Sparc business, headquartered in Palo Alto, Calif., had studied ways to assist Sparc processors in running x86 software by emulation. He hired Colin Hunter as a short-term contractor on a project to determine what new instructions might be added to Sparc to help make emulation run faster. They completed the project and produced an internal report. But it appeared unlikely that merely adding a few new instructions to Sparc would significantly enhance the processor's ability to run x86 software.

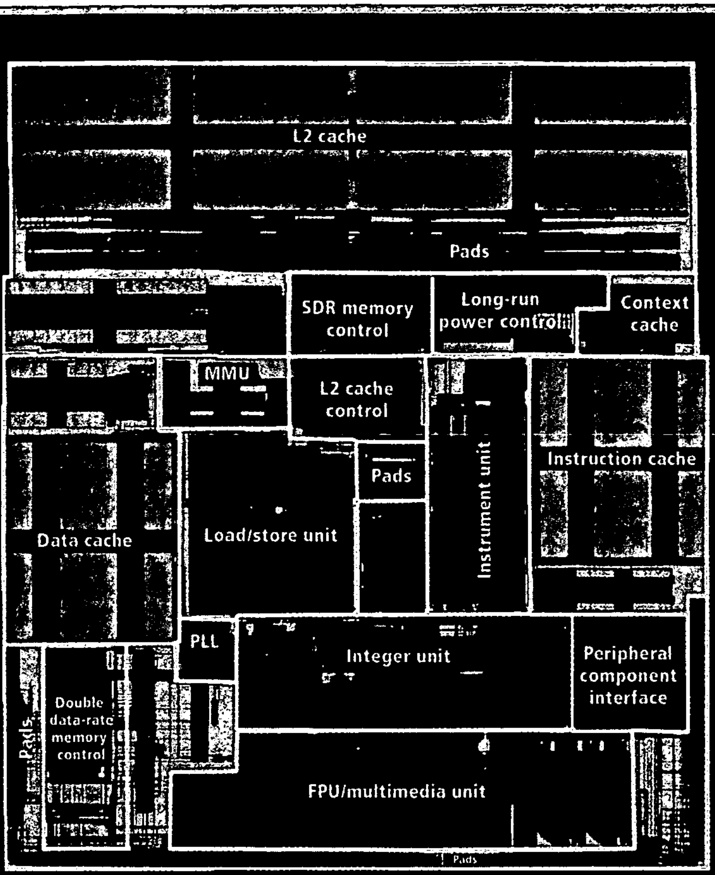
Ditzel had also become concerned about the ever-growing complexity of microprocessor design. He had long been a champion of simple microprocessors: with a professor from the University of California at Berkeley, David Patterson, he had coauthored the pioneering 1980 paper "The Case for the Reduced Instruction Set Computer." But as time went on, he told *IEEE Spectrum*, more and more functions got piled into RISC chips.

This complexity meant that RISC chips were getting bigger and hotter and were taking much longer to design and debug, and improvements in performance were limited. Some chip designs were so complex, in fact, that hundreds of engineers were needed for one design team. Looking out 10 years into the future, Ditzel thought things would only get worse.

So, in early March 1995, he quit his job at Sun. Within a few weeks, he had an idea worked out for a new type of microprocessor. The new device would be fast and simple, and although it would bear no resemblance to an x86 processor, it would be surrounded by a layer of software that could transform, on the fly, an x86 program into code that the simple microprocessor could understand. The technique, called dynamic binary translation, gives programs the impression that they are running on an x86 machine.

Ditzel called on Colin Hunter again and the two prepared to file papers to incorporate as a company. But first they needed a name, one that would not give away what they were doing and one not already taken by any of the other numerous technology companies in California. After running various combinations of high-tech sounding syllables past the California Secretary of State's office, they found one that was available—Trans-

9



[1] The highest-performance Crusoe chip, the TM5400, is 73 mm^2 in area. It contains a 128KB level one (L1) cache and a 256KB level two (L2) cache, on-chip LongRun power control, an integrated peripheral-component-interface bus and double- and standard-data-rate dynamic RAM controllers.

The chips were fabricated by IBM Corp. at its foundry in Burlington, Vt. They feature five levels of copper interconnect and a minimum feature size of 0.18 μm.

(ILLUSTRATIONS: TRANSMETA CORP.)

meta. "We thought we'd change it later," Ditzel said, "but now that it has so much recognition, we'll keep it."

Ditzel and Hunter started making the rounds at various venture capital companies. Meanwhile, the team grew.

The two were joined first by Steve Goldstein, a former vice president of sales and marketing at Ross Technology Inc. (which closed in 1998). Also signing on was a group of Sun engineers who had also been struggling with the problem of how to create a fast emulator: Doug Laird, now senior vice president of product development, Greg Zyner, a very large-scale integrated chip designer, Malcolm Wing, a chip architect, Ed Kelly, a systems engineer, and Bob Cmelik, a software engineer.

The company set up shop in Ditzel's Los Altos Hills house, taking over the living room and two spare bedrooms and equipping them with Sparcstation computers, PCs, printers, an overhead projector, a fax machine, a copier, whiteboards, and obligatory munchies. The team met there several afternoons every week.

Meanwhile, Laird and Zyner set up camp in the living room of Laird's Los Gatos ranch house to sketch out the chip design on a whiteboard commandeered from Laird's five-year-old daughter. Laird and Zyner would visit Wing's Menlo Park apartment, where Wing was working on the overall chip architecture and working with Cmelik on hardware and software tradeoffs.

By summer 1995, funding was getting to be a big concern. Transmeta learned that the money supposedly promised by a venture capital firm was to come from a new fund that had not actually been financed. To get their hands on some money quickly, Laird and Ditzel took a contract from the Advanced Research Projects Agency (ARPA) (now DARPA), Arlington, Va., to write several white papers about high-speed CMOS design techniques. They received $250 000 for this work. The proceeds were used to pay salaries to several members of the group, and to rent a real office building in Redwood Shores, Calif.

MAKING PROGRESS

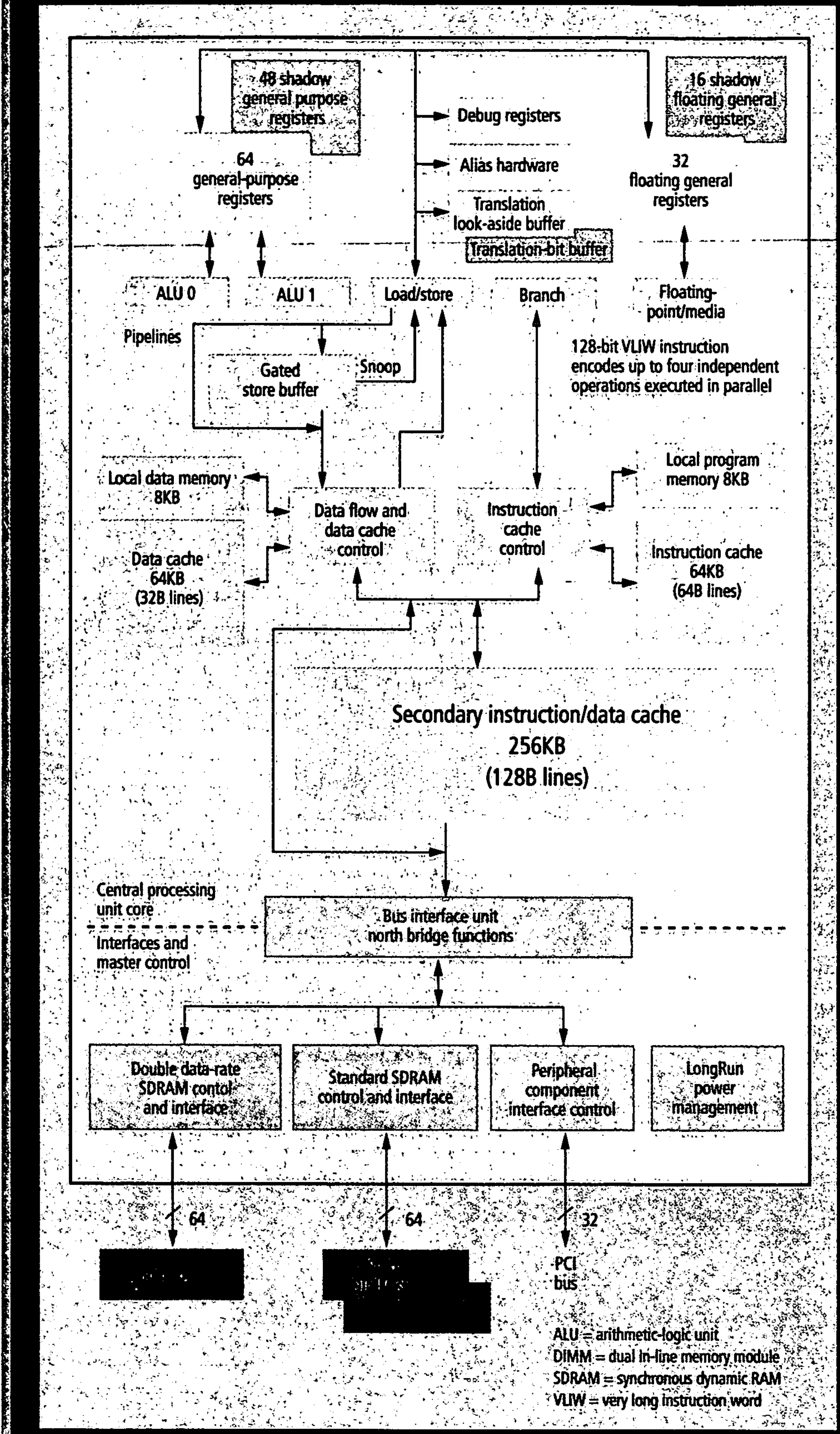
Despite the engineers' worries over financing, the technical work on the new microprocessor was proceeding, and some key breakthroughs had been made.

For one, Ditzel chose to base the chip's design on a well-known technique called very long instruction word (VLIW) [Fig. 2]. The attraction of a VLIW microprocessor was the simplicity of its design and its high performance.

A growing difficulty with other commercial architectures, both RISC and x86, stemmed from a common method of improving performance, namely issuing multiple instructions per clock, a technique called superscalar execution. In RISC and x86 superscalar designs, scheduling the instruction order and determining which instructions can be executed at the same time is left to the microprocessor hardware. This setup greatly complicates the design of these systems, slowing them down, adding cost, and burning power. As designers add more and more execution units to the chip in their search for better performance, a point of diminishing returns is reached when gains are largely eroded by the added complexity.

VLIW processors also execute many instructions in parallel (the Transmeta chips can execute four), but it is the job of a compiler (read software) to schedule the instructions. This also fits in with Transmeta's scheme of assigning more work to the software. In Transmeta-ese, individual instructions are called atoms and the VLIW instruction groups are called molecules. The company designed its final chip so that the atoms arrive at the processor bundled by the compiler into molecules composed of two or four

9


48 shadow general purpose registers
Debug registers
16 shadow floating general registers
64 general-purpose registers
Alias hardware
32 floating general registers
Translation look-aside buffer
Translation-bit buffer
ALU 0
ALU 1
Load/store
Branch
Floating-point/media
Pipelines
Gated store buffer
Snoop
128-bit VLIW instruction encodes up to four independent operations executed in parallel
Local data memory 8KB
Data flow and data cache control
Instruction cache control
Local program memory 8KB
Data cache 64KB (32B lines)
Instruction cache 64KB (64B lines)
Secondary instruction/data cache 256KB (128B lines)
Central processing unit core
Interfaces and master control
Bus interface unit north bridge functions
Double data-rate SDRAM contol and interface
Standard SDRAM control and interface
Peripheral component interface control
LongRun power management
64
64
32
PCI bus
ALU = arithmetic-logic unit
DIMM = dual in-line memory module
SDRAM = synchronous dynamic RAM
VLIW = very long instruction word

[2] Very-long-instruction word (VLIW) architecture forms the basis of the Crusoe chips. It consists primarily of working general-purpose and floating-point registers and their shadow registers, the executions units, memory, and memory control circuits. The chips have a built-in north bridge, and the TM5400 features LongRun power-management circuitry.

atoms that can be processed together, and the processor executes them.

Another early breakthrough was understanding the factors that traditionally made emulation slow and developing alternatives to eliminate these obstacles. A key reason for the sluggish performance was the extra instructions that an emulator has to run to match the exact state of a processor in a different architecture. "In traditional emulation," Laird told *Spectrum*, "you are taking a program written for a processor with one architecture and getting it to run on a processor with a different one, and the states of the two processors are not the same."

For instance, an x86 program may expect a processor to set a condition code, and the program performs a branch operation based on the value of that condition code. But when the program is run on a PowerPC, say, the condition code is not generated in the same way that an x86 processor would have generated it. So the emulator has to go through a number of PowerPC instructions to set the condition code in the same way as the x86.

"What we discovered," said Laird, "was that if you can facilitate implementing the state of the first processor in the second one by designing certain registers to hold that state, the emulation software doesn't have as big an overhead."

Another difficulty about emulation has to do with so-called exceptions, which are caused by processor faults, errors, traps, or other exceptional events. Since exceptions halt the execution of a program, the operating system must find the cause of the exception and re-execute the instructions that faulted in a way that isolates the fault. The question of how to deal with exceptions was brought up early in the design process. It was Cmelik who identified the seriousness of the problem—not solving it would mean a dead-end for the technological approach being taken.

The problem arises, explained Laird, because the VLIW program they created reorders the x86 instructions. So if the x86 program creates a fault, such as a divide-by-zero—although it may happen infrequently, it still may happen—the processor has to be able to create the exact same state as any other x86 processor would, and hand it off to the operating system to deal with the fault.

The solution came several weeks later with a novel hardware/software combination called commit and rollback, which, according to Wing, "is really the fundamentally different thing about our machine."

Commit and rollback was implemented by creating an extra set of registers, called shadow registers, in addition to the working registers. With the execution of a software commit instruction, the shadow registers duplicate the data in those working registers. As the operation progresses, the working registers are updated by each computational operation. But the shadow registers are not updated until the processor receives an all-clear signal in the form of another commit instruction, indicating that no exception occurred.

When the processor hits a fault, Transmeta's software issues a rollback instruction, and the information in the shadow registers is copied back into the working registers. "So we can reverse the execution," said Laird. "You come to a state, say, 'Oops, I did a bad thing,' go back in time instantly in one cycle, and start again." The next time around, the software schedules the operations more conservatively, say, by executing the instructions in precisely the same order as the original x86 program.

The team realized that, in the case of a rollback, data to be stored in memory would also have to be rolled back. They came up with a circuit called a gated store buffer to keep track of the stores between commit points. If an exception occurs in this period, the system can instantly roll back to the previous state and discard those stores.

The gated store buffer has a committed and an uncommitted side with a "gate" in the middle. After some compilation creates the data to be stored, the data goes to the uncommitted side of the buffer. After a commit instruction, the gate opens and the data on the uncommitted side moves to the committed side and is then stored in memory.

This process may involve a substantial amount of data. A single x86 instruction, for example, can modify 130 bytes of memory. Other superscalar microprocessors also need store buffers, but nothing quite so big.

IT'S CODE MORPHING!

While development of the chip architecture was progressing, it was beginning to look as if the group might never get funded. Group members kept explaining to venture capitalists that with their revolutionary software-based microprocessor, they could attack markets previously owned by x86 chips, but no one bit. By the end of the summer of 1995, Ditzel and Hunter had pitched nearly 30 venture capitalists; Laird often went along as an observer.

"They just didn't get it," Laird said. "Dave [Ditzel] would start talking about dynamic binary translation, and their eyes would just glaze over. We were pumped up, saying this is a great idea, it is a new microprocessor, and nobody has ever done it this way, but we could've been from Mars for all they cared. We were just getting too technical."

"It was a hard sell," Ditzel told *Spectrum*. "We were saying we wanted to do hard core R&D and develop this big new idea and it would take four years. And the venture capitalists would say, 'Couldn't you just have a simple idea you could do in six months?' "

So in midsummer the entire team sat down at their new offices in Redwood Shores to figure out another way to pitch their ideas. They concluded that they needed to sum up the essence of what they were doing in a word or two, a simple, catchy name that the venture capitalists would understand. After tossing around several ideas, Cmelik threw out the term "Code Morphing" and they knew they had it.

They also discarded some of their more technical PowerPoint slides and came up with a simple sketch of their concept, which they called the amoeba [Fig 3]. The amoeba explained how a traditional microprocessor was, in their design, to be divided up into hardware and software.

Ditzel went back to the venture capital community with the new pitch. Laird sat on the sidelines with his watch. "I timed how long it took, from the first time Dave said Code Morphing, to the time the venture capitalists started using the word themselves," Laird said. "It was less than 5 minutes."

Within a few weeks, several venture capital firms were competing to fund the group. By October they had commitments from Institutional Venture Partners, Menlo Park, Calif., and Walden Group, San Francisco. The check for $3.5 million arrived in December 1995.

"We hadn't changed the principles, we hadn't changed who we are, we hadn't changed anything except how we presented it," Laird said. "We said 'Code Morphing software' and snap, we got funding."

Since trademarked, the buzzword aptly describes what the software does: it takes x86 instructions and recompiles them on the fly into VLIW instructions. As it recompiles them, it optimizes them, making them run, in many cases, more efficiently than the original x86 code. What happens with x86 applications is that, in the rush to market faced by software writers, often applications are compiled without the highest levels of compiler optimization to facilitate debugging. Once the software works, it is shipped; there is no time in the schedule to go back and recompile and re-test, meaning that many software applications have room for improvement.

On a typical software application program, such as Microsoft Word, Code Morphing works like this: it starts with the x86 binary code for a program section to, for example, edit text. In real time, the code goes into Transmeta's software and comes out the other side transformed into VLIW code. In the software's sequence of operations, the x86 instructions are first translated into a sequence of VLIW atoms. Then an optimizer, using some new and some well-known compiler techniques, checks to see if the code can be improved—for instance, by the elimination of redundant atoms.

Finally, a scheduler reorders the atoms and groups them into molecules [Fig. 4]. Once translated, the VLIW code is stored in a special part of memory, accessible only by the Code Morphing software, so that particular program need not be translated again.

But that is not the end. The new software continues to monitor how an application is being used. If it finds that a user is spending a lot of time changing the font, for instance, it turns on more levels of optimization to make that part of the program run faster. "We only optimize that portion of the code [being used]," explained Laird. "For the things that are executed infrequently, there is no reason to put in that overhead."

One of the challenges of creating the Code Morphing software was to make the Crusoe processor, in many cases, bug-compatible with the x86 so that it would generate the so-called Blue Screen of Death at many of the same times an x86 processor would.

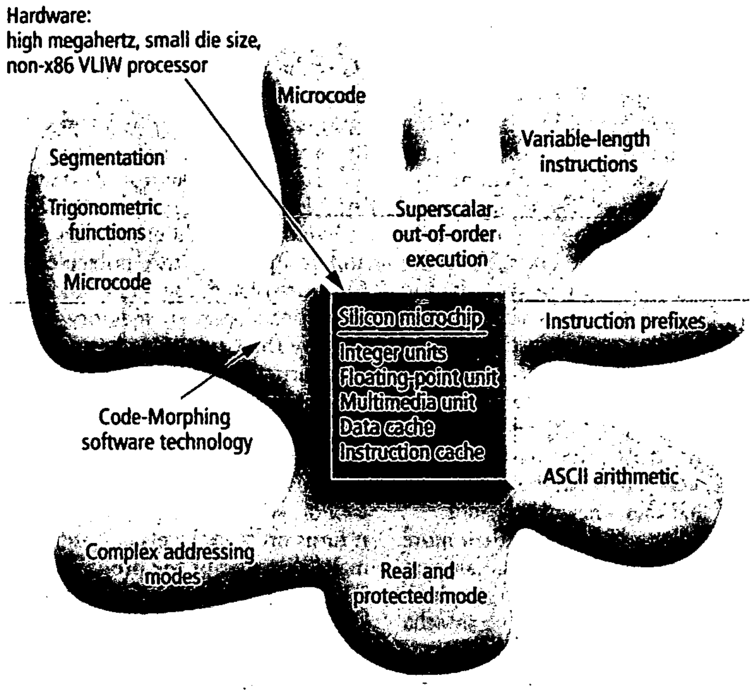
A REAL COMPANY

Now that the funding was in place, it was time in late 1995 to build this small team of engineers into a real company and actually implement the new microprocessor architecture on a chip.

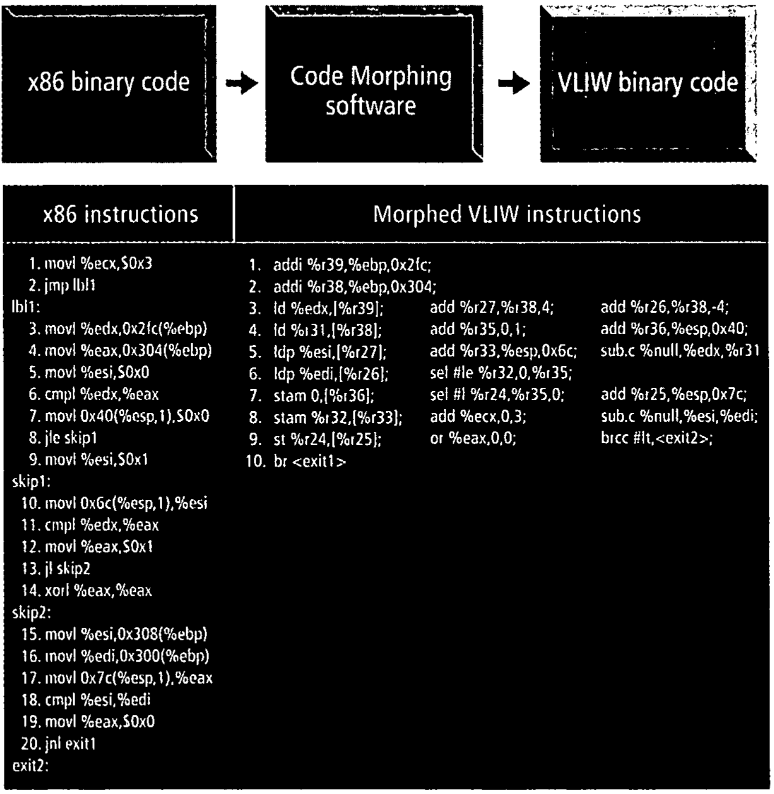
The design the team came up with contained only about half the logic transistors of an x86 processor. It included five execution units—two arithmetic-logic, a load/store, a branch, and a floating-point—and it could execute four instructions in a cycle. Sixty-four general-purpose and 32 floating-point working registers were shadowed by 48 general-purpose and 16 floating-point registers. Memory, memory management, and the so-called north bridge (usually a separate IC) rounded out the design.

Even more important was what the design did not include. It had no superscalar decode, grouping, or issue logic. It had no register renaming or segmentation hardware. And it had no floating-point stack hardware. Nor did it have memory management in the front end of the machine. It also had less interlock and bypassing logic than a traditional central processing unit. This structure contributed to a simpler design with far fewer transistors, which was the key to low power.

In late 1995, Transmeta started hiring engineers to join the eight founders and begin mapping out details of the architecture. The first few hires were people whom



[3] The Transmeta founders credit a simplified sketch of their proposed architecture, which they called the "amoeba," with convincing the financial community that their idea could work. In this concept, the x86 architecture is an ill-defined amoeba containing such features as segmentation, ASCII arithmetic, and variable-length instructions; the square inside the blob is the proposed VLIW processor and its functions.



x86 instructions	Morphed VLIW instructions

```
   1. movl %ecx,$0x3
   2. jmp lbl1
lbl1:
   3. movl %edx,0x2fc(%ebp)
   4. movl %eax,0x304(%ebp)
   5. movl %esi,$0x0
   6. cmpl %edx,%eax
   7. movl 0x40(%esp,1),$0x0
   8. jle skip1
   9. movl %esi,$0x1
skip1:
  10. movl 0x6c(%esp,1),%esi
  11. cmpl %edx,%eax
  12. movl %eax,$0x1
  13. jl skip2
  14. xorl %eax,%eax
skip2:
  15. movl %esi,0x308(%ebp)
  16. movl %edi,0x300(%ebp)
  17. movl 0x7c(%esp,1),%eax
  18. cmpl %esi,%edi
  19. movl %eax,$0x0
  20. jnl exit1
exit2:
```
	```
	 1.  addi %r39,%ebp,0x2fc;
	 2.  addi %r38,%ebp,0x304;
	 3.  ld %edx,[%r39];     add %r27,%r38,4;      add %r26,%r38,-4;
	 4.  ld %r31,[%r38];     add %r35,0,1;         add %r36,%esp,0x40;
	 5.  ldp %esi,[%r27];    add %r33,%esp,0x6c;   sub.c %null,%edx,%r31
	 6.  ldp %edi,[%r26];    sel #le %r32,0,%r35;
	 7.  stam 0,[%r36];      sel #l %r24,%r35,0;   add %r25,%esp,0x7c;
	 8.  stam %r32,[%r33];   add %ecx,0,3;         sub.c %null,%esi,%edi;
	 9.  st %r24,[%r25];     or %eax,0,0;          brcc #lt,<exit2>;
	10.  br <exit1>
	```

VLIW = vey long instruction word

[4] Code Morphing software transforms x86 binary code into individual instructions, called atoms, for the Crusoe processors. The compiler within the new software then looks for atoms that can be executed together and groups them into very long instruction words called molecules. Molecules may contain two or four atoms. In cases where an atom is to be executed alone, or only three atoms are to be executed together, the second or fourth position is filled by a no-operation instruction.

Laird, Hunter, or Ditzel had known for years, starting with Godfrey D'Souza, a Sun engineer who would have been in the founding group had he been in a financial position to work without a salary. In 1996, some 80 more engineers were added, mostly mid-career engineers who had years of experience in the jobs they were to take on for Transmeta.

Signing on so many experienced engineers so fast in Silicon Valley's tight job market turned out to be surprisingly easy.

"My being old helped," Laird said. (He is 44.) "I've been around a long time; I know a lot of people."

Ditzel also had a lot of contacts. "I had worked at Bell Labs," he told *Spectrum*, "and when you work there, you tend to get invited to lots of places to see their secret projects. I had been doing a lot of work for IEEE and ACM [Association for Computing Machinery] on conferences, and I had gone to school with people who had gone on to be professors at universities. So I was able to just pick up the phone and call the right people."

When Ditzel and Laird made such calls, they provided little information to their prospective hires—just that they had a new company and were doing something really cool and new in computer architecture. After they were sure the person was interested—and was the right fit—they brought out a nondisclosure agreement. Only after it was signed did they reveal any details about their plans.

The experience of Guillermo Rozas was typical. Rozas, a software engineer and now Transmeta's director of product development, was at Hewlett-Packard Laboratories, in Palo Alto, in 1997 when he heard from a close friend who had signed on with Transmeta. As Rozas explained, "He was a really smart guy, and he told me there were really smart people here that would be fun to work with. I didn't know all that much more when I came in, other than a lot of people I had known had mysteriously disappeared inside Transmeta."

Also recruited was Stephen Herrod, now director of software productization, who was at Stanford University, California, before joining Transmeta. He had done his Ph.D. dissertation on runtime code generation, citing a number of papers and researchers in the field. "When I searched out where all those people were now, it turned out that all of them were at Transmeta," he told *Spectrum*. "I did know someone here from conferences, so I called him up and asked if I could come in. I was about the 15th software person hired, and the other 14 were largely the people whose work I had been studying."

In late 1996, after some hundred people were on board, Laird decided it was time to hire a few engineers right out of college.

G

"You need a good distribution of experience," he said. "If you have all senior level people, and there are a lot of details that need to be taken care of, they are not going to want to do that." He and Ditzel called their professor friends and asked for their best students, eventually hiring around 30 graduates. A number of these students were interviewed without even knowing what Transmeta did, only that their advisor had told them that Transmeta was a hot start-up.

Despite the large numbers of engineers that were being hired from Silicon Valley's top companies–Hewlett-Packard, MIPS Technologies, Silicon Graphics (but not Intel)—little information about Transmeta's work was being leaked.

"Our approach was simple: to use software as a key piece of the microprocessor," Ditzel said. "So if that one simple idea leaked out, our competitors could get a project going. If it didn't, then they couldn't have a competitive product out in five weeks—it would take them five years."

They kept the secret virtually leak-free by what Ditzel calls rifle-shooting. "Leaks come from people you interview and don't hire. But if you rifle-shoot the exact people you want, all you have to do is impress them about what you're doing and hire them. Then once they've joined your company, they won't leak." He says some 90 percent of engineers offered jobs by Transmeta accepted.

"People were excited about this project because it was one of the first really different types of computer systems that had been designed in the past several years," Ditzel told *Spectrum*. "The hardware guys loved it because they could start with a blank sheet of paper, they didn't have to be compatible with an old instruction set. The software guys liked it because they could ask the hardware guys for special features."

Because the company was hiring so many senior people, the decision was made in the beginning that, even though funds were tight, every engineer would have a private office (as soon as they were available—some employees did double up temporarily). Other amenities include a well-stocked kitchen with drinks, sandwich makings, and snacks. Dinner is ordered in four nights a week.

The atmosphere is as open as a college campus (complete with a busy foosball table)—perhaps even more so. Said Keith Klayman, a member of the technical staff: "Like at a university, we can go to anyone here if we have a question. But at the university, the professor was in maybe once a week. Here, the high-level people are always around and accessible."

Every engineer also has at home a company-provided computer that connects to the Internet through a high-speed digital subscriber line (DSL). With this equipment, people with families can go home for dinner, get back to their engineering work around 10 p.m., and then sleep late in the morning. One winter the company even rented a cabin in the Lake Tahoe ski area and equipped it with computers and DSL capability, so engineers could get their winter skiing in without losing time from their projects.

The lack of borders between hardware and software engineers at Transmeta is, employees report, unique in their experience. Whenever a technical problem is discussed, both hardware and software engineers team up to address it. Sometimes a problem faced by the software engineers is made solvable by a change in the hardware; sometimes it goes the other way. As a result, the company's fleet of rattletrap bicycles, used by the engineers to travel between the buildings housing the two teams, get a lot of use [Fig. 5].

HOUSTON, WE HAVE A PROBLEM...

After three years of work, in August 1998, the first chips came back from IBM Corp., which had signed on as manufacturer. To check out the performance of the chips, the Transmeta engineers ran several benchmarks, both for Unix and Windows. The chips ran Unix benchmarks as fast as had been expected; the first magic trick had worked.

But when the engineers assigned to performance analysis started testing Windows benchmarks, they had a nasty surprise. The Windows benchmarks reported scores far lower than expected. Transmeta had reached into its magic hat to pull out a rabbit and had instead come up with a turtle.

"It was like in the *Apollo 13* movie," Laird said, "We wanted to say, 'Whoops, Houston, we've got a problem here.' "

Laird was philosophical about the situation. "We're engineers," he told *Spectrum*. "We didn't need to panic. We needed to understand what was going on. And so we analyzed it, moved teams of hardware and software people onto it, and started fixing it."

But not all the engineers at Transmeta were so sanguine.

"We had been riding high, blindly expecting the chips to do everything that we had promised," recalled Klayman. "When they didn't, it was a real morale killer." Some of them felt it was never going to work, and since nobody was motivated, no work was getting done. Then Doug Laird told them to drop everything else they were doing, as there was still a chance to right the ship.

The company held an all-hands meeting, in which Laird told everyone the truth—that they had run into a wall running Windows benchmarks. But he reassured them that, by working together, they could fix the problem. Murray Goldman, a member of the board of directors, pledged that the board would stand by their efforts, implying that more money would be raised, should it be needed.

Looking back, Laird said a problem might have been expected with Windows95 applications. "Most of us came from a Unix background, we knew how Unix applications behaved. But we didn't really understand Windows95," he said.

Apparently Windows95 still had a lot of old 16-bit code in it, whereas Unix (as well as Windows NT) used a flat memory model with pure 32-bit code. Supporting 16-bit code was something that Transmeta had decided to offload into software.

Once they realized this, they redesigned the hardware to give better support to Windows95 applications. They also increased the size of the caches because Windows95 applications tend to use more memory than Unix applications.

The redesign process added about a year to Transmeta's development time. In fact, getting products to market took longer than any of the founders had anticipated. "If we had had a better idea of how long it would have taken, we probably would not have done it, I suspect," said D'Souza.

TO MARKET, TO MARKET

While the engineers were struggling to redesign the chip to run Windows applications at a reasonable speed, a marketing team was taking the show on the road, showing off their concept to OEMs, and asking them if Transmeta was making chips that would sell, and, if so, into what market.

The feedback from the OEMs was almost unanimous, Ditzel said. While they had been presenting their product as appropriate for both the desktop and mobile markets, customers disliked the split focus. They wanted chips optimized for mobile computing.

"Customers told us consistently," Ditzel said, "that they had pretty good chips for desktops and servers, but the road ahead for mobile chips looked horrible; there was nothing coming out that was usable. So, they told us, if you are going to build us a chip, go build us a mobile chip."

The most important parameter for the mobile market is a chip's power consumption. Ditzel said he and Laird had always thought that the hardware/software architecture had a lot of potential for reducing a chip's power consumption, and in general the team designed the chip's circuits with low power in mind. They had not pitched this feature to venture capitalists, because, Ditzel said, it was impossible to know how significant the drop in power was going to be.

By late 1998, with the initial market research complete and prototype chips on which to measure power consumption in

[5] Transmeta engineers built a PC-compatible board so they could run Windows applications on their chips. The board is held at left by Doug Laird, vice president of product development.

Transmeta's hardware and software teams are housed in two buildings, about 1 km apart in an office park. Designing chips with so many software functions requires interdisciplinary teams, so Transmeta's fleet of secondhand bicycles gets a lot of use in cross-campus commutes—even by chief executive officer David Ditzel [below].

PHOTOS: DAVID TEORGE/BLACKSTAR

hand, the decision to focus on mobile computing was made, and power consumption issues came to the forefront.

POWERING DOWN

"A number of people have said that designing lower-power chips means doing a lot of little things—a little bit here, a little bit there," Laird told *Spectrum*. "And if you do a lot of it, the sum of it is good."

One of the biggest little things that the Transmeta team did was to offload a good bit of the microprocessor function onto the software, which allowed them to design simple streamlined hardware with about half the number of transistors of an x86 chip. "Obviously," continued Laird, "if you have fewer transistors, you burn less power."

The team also used virtual devices to cut down on the amount of hardware. A virtual device is one that is not exactly the same as the device expected by the program, but produces the same result. It works by using the Code Morphing software to monitor the input and output instructions to the device, then to send those instructions to the virtual device instead. For example, Crusoe incorporates, on-chip, a separate IC called the north bridge, which couples the processor to the peripheral component interface bus and to external memory.

The north bridge features architecturally defined registers, to which the program sends input and output instructions. To be compatible with the architecture for which the instructions are written, those registers must be constructed so that any application, or the operating system, can manipulate them correctly.

But rather than implementing those registers exactly as in a conventional north bridge, Transmeta engineers employed the Code Morphing software to intercept the instruction to the north bridge registers and send it instead to the registers defined in the Crusoe architecture. Ditzel predicts that the team will be virtualizing more circuits as time goes on.

Another technique is to turn on only those functional units that are absolutely needed to execute an instruction. The process requires a separate clock for each combination of functional units that is turned on during the execution of an instruction. This approach was carried out so thoroughly that a vendor supplying a computer-aided design simulation tool complained that the Transmeta design "broke his tool" because the processor had over 10 000 clocks to control which units get turned on, and when.

But the biggest breakthrough in low-power design came with the development of the so-called LongRun technology, which uses the Code Morphing software to monitor applications as they are running. Then LongRun hardware adjusts both the supply voltage and the clock frequency so that each application runs only as fast as it must to get the job done. Since the processor is running at maximum efficiency, it is maximizing battery life.

Traditional power-management systems also adjust power, but are much less refined. They often try to extend battery life by varying the duty cycle, repeatedly turning the central processing unit on for a fraction of a second, then off for a fraction of a second. "Imagine that you wanted to make the light in a room half as bright," explained Marc Fleischmann, manager of the LongRun power management team. "It would seem silly to do that by flipping the light switch on and off rapidly. But that's exactly how power management works on traditional notebook computers."

Rather than a light switch, Fleischmann compares LongRun to a dimmer control. While applications are running, Transmeta's software observes the traditional power management states and the time spent in the sleep mode; then on-chip LongRun circuitry reduces the frequency and the voltage to precisely match just what the user needs.

"If you spend 40 percent of your time in sleep mode, that means you only need to run at 60 percent of the performance level. So we reduce the frequency from 700 MHz to about 400 MHz, say. And we ramp down the voltage correspondingly. Adjusting both frequency and voltage is a far more efficient way to extend battery life," Fleischmann told *Spectrum*.

"The major point," added Laird, "is that

9

LongRun is an extension of power management, not a substitution for it."

All told, the efforts to reduce power consumption on the Crusoe chips can reduce power by a factor between three and 30, depending on the application, compared with a typical x86 processor, according to Fleischmann.

SOFTWARE'S EDGE

As the design of the microprocessor evolved, other advantages of moving functions into software became apparent. "Having software involved gave us more opportunities than we initially thought," Ditzel said.

Processor upgrades are simplified because the layer of software between the applications and the chip frees the designers to change the chip architecture without causing x86 software developers to have to recompile their code. Code Morphing software can be updated independently of hardware by loading a software upgrade into Flash memory.

The software also helps the debugging process. When the hardware design team got the very first silicon, they found plenty of bugs. They knew that the software layer would help them debug the chip, but no one appreciated ahead of time just how powerful that help would be, according to D'Souza. They were able to work around a lot of the bugs, he said, by performing operations in a different way.

The engineers were always able to boot Windows, even on buggy silicon. As each bug was found (and fixed with software), it was added to the list of revisions for the next design.

What's more, the software layer was also used to increase performance by improving the timing of critical paths. For instance, engineers found that when two particular atoms were paired together in a molecule, the processor ran sluggishly. Otherwise, the chip could run at a much faster clip. So the hardware designers asked the software designers to modify the scheduler so that these two atoms would not appear in the same molecule. "All of a sudden," said D'Souza, "we were running at 600 MHz instead of 466 MHz."

CRUSOE LIVES

By August 1999, the first of the redesigned chips came back from the IBM fab. This time, it ran Windows applications just fine. This chip, for the mobile computing market, became the TM5400. The original design, which was intended for running Linux for the Internet appliance market, became the TM3120.

The TM5400 is similar to the TM3120, but has added the LongRun feature to conserve power. This chip also has more on-chip cache memory than the TM3120 to support x86 applications for Windows-based notebook computers. The TM3120 runs at 400 MHz, while the TM5400 runs at up to 700 MHz.

Transmeta engineers intentionally designed Crusoe to be simpler than conventional x86s slated for mobile applications, but to achieve comparable performance by running at a higher frequency. The fastest mobile Pentium III clocks in at 650 MHz.

Of course, the performance of the Crusoe chips depends on the application. "I think it's fair to say that Crusoe is faster on some applications and not as fast on others," said Ditzel.

For most mobile applications, all of the TM5400's processing power is often not even needed. The effectiveness of LongRun lies in making the processor run at just the right frequency to deliver the performance demanded by the application while conserving power.

The microprocessor family was formally branded Crusoe, after the fictional adventurer and traveler, Robinson Crusoe. "It was friendly, short, and easy to remember," Ditzel said. "So you'll remember it's a mobile chip."

Finally, on 19 January 2000, after nearly five years of effort and over $100 million invested, Transmeta pulled back its curtain at a large press conference at Villa Montalvo, a grand old estate in the hills of Saratoga, Calif.

Meanwhile, engineers at one of Transmeta's unmarked buildings raised a huge black flag with the yellow Crusoe logo from the roof of their building. The flag could be seen by Intel engineers driving to and from their nearby offices.

Bennett Smith, a consultant in microarchitecture, computing platforms, and related intellectual property, is impressed by Transmeta's technology. "They have a sophisticated approach to power consumption that looks pretty amazing," he told *Spectrum*. On the negative side, he has heard concerns that the company's chips are just too expensive. "Companies designing for the portable market may have difficulty justifying the intellectual property premiums built into Transmeta's business plan," he said. Smith and Bruce Shriver are co-authors of *The Anatomy of a High-Performance Microprocessor: A Systems Perspective* (IEEE Computer Society Press, Los Alamitos, Calif., 1998).

Writing in *Cabners Microprocessor Report*, 14 February 2000, Tom R. Halfhill also expressed cautious praise: "Revolutionary may be an overstatement, but they are definitely different....The TM5400's LongRun feature is one of the most innovative technologies introduced by Transmeta. To our knowledge, no other microprocessors can conserve power by scaling its voltage and clock frequency in response to the variable demands of software."

Indeed, the chips still continue to amaze their creators.

Referring to the prototype system that the Transmeta team used to test the Crusoe chips, Rozas said, "I've been seeing these things run now for a year and a half. I know them inside out. Yet, I am still amazed every time I start it up and [a Crusoe-chip computer] looks like a normal PC.

"Considering the complexity of the project, it is amazing how well it works, how fast it works, and how low-power it is," Fleischmann commented. "For the end-user, this is just a normal PC, but under the hood, it is a technological marvel. I am in a state of wonder, too—and I am proud."

THE NEXT GENERATION

Variations of the current generation (both low-cost versions and higher-performance versions) are also being designed. (The part numbers were purposely picked to be in the middle of the range, leaving room for both new versions.)

Transmeta's next generation may have a fundamentally different architecture, even a different instruction set—whatever it takes to make it better, because use of Code Morphing software obviates the need for legacy hardware.

The design will most likely include the latest submicron CMOS technology, including shielded clock lines. The computer-aided design tools will need to make accurate models of inductive coupling between the interconnect structures on the chip. To the engineers, this is a chance, once again, to start with a blank sheet of paper and to rethink the first generation's tradeoffs between hardware and software. To the user, though, the next Crusoe will still appear as an x86.

"Usually you say the next generation will be bigger and better," Ditzel said. "But in this case, I'll say it will be smaller and require even less power." ◆

TO PROBE FURTHER

Information on Transmeta Corp., white papers describing in detail its Code Morphing technology, videos of the Crusoe product launch event, recent news articles, and employment opportunities at the company are available at www.transmeta.com.

Detailed analysis of the Crusoe processor architecture is to be found in the article "Transmeta breaks x86 low power barrier," by Tom R. Halfhill, *Microprocessor Report*, 14 February 2000, p. 1 and pp. 9–18.

Transmeta will be making presentations at the Embedded Processor Forum in June (see www.mdronline.com) and at the IEEE's Hot Chips meeting in August in California (see www.hotchips.org).